

# INTEROPERATION FOR INCOMPATIBLE EVALUATION STRATEGIES

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

William Faight

June 2008

## AUTHORIZATION FOR REPRODUCTION OF MASTER'S THESIS

I reserve the reproduction rights of this thesis for a period of seven years from the date of submission. I waive reproduction rights after the time span has expired.

---

Signature

---

Date

## APPROVAL PAGE

TITLE: Interoperation for Incompatible Evaluation Strategies

AUTHOR: William Faight

DATE SUBMITTED: June 2008

Dr. John Clements

Advisor or Committee Chair

\_\_\_\_\_

Signature

Dr. Gene Fisher

Committee Member

\_\_\_\_\_

Signature

Dr. Aaron Keen

Committee Member

\_\_\_\_\_

Signature

## **Abstract**

### Interoperation for Incompatible Evaluation Strategies

by

William Faight

Software components written in different programming languages can cooperate through interoperation. Differences between languages—incompatibilities—complicate interoperation. This paper explores and resolves incompatible type systems, support for parametricity, and evaluation strategies with a model of computation, gives a thorough proof of its type soundness, and describes an implementation of it. The model uses contracts for higher-order functions and lump types to resolve incompatible type systems, label types to resolve incompatible support for parametricity, and delayed conversions for list constructions to resolve incompatible evaluation strategies. These mechanisms enable the interoperation of Haskell, ML, and Scheme without compromising their semantics.

## Acknowledgements

I want to thank my parents, Jerry and Jo Ann, for their encouragement, advice, and support, without which this would not have been possible.

I want to thank my adviser, John Clements, for helping me along the way. I very much appreciate the time he set aside for me and his advice.

# Contents

<b>List of Figures</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Model of Computation</b>	<b>4</b>
2.1 Grammars . . . . .	5
2.2 Typing Rules . . . . .	7
2.3 Operational Semantics . . . . .	8
2.4 Interoperation Models . . . . .	8
2.4.1 Natural Number Types . . . . .	9
2.4.2 List Types . . . . .	9
2.4.3 Function Types . . . . .	11
2.4.4 Forall Types . . . . .	12
<b>3 Proof of Type Soundness</b>	<b>34</b>
3.1 Proof of Expression Progress . . . . .	34
3.2 Proof of Type Preservation . . . . .	45
<b>4 Related Work</b>	<b>57</b>
<b>5 Future Work</b>	<b>60</b>
<b>6 Conclusion</b>	<b>61</b>

# List of Figures

2.1	Conversion of a function . . . . .	12
2.2	Conversion of a higher-order function . . . . .	13
2.3	Labels protect parametricity . . . . .	14
2.4	Labels detect parametricity violations . . . . .	15
2.5	Polymorphic function converted to Scheme function . . . . .	17
2.6	Haskell grammar and evaluation contexts . . . . .	18
2.7	Haskell typing rules . . . . .	19
2.8	Haskell operational semantics . . . . .	20
2.9	Haskell-ML operational semantics . . . . .	21
2.10	Haskell-Scheme operational semantics . . . . .	22
2.11	ML grammar and evaluation contexts . . . . .	23
2.12	ML typing rules . . . . .	24
2.13	ML operational semantics . . . . .	25
2.14	ML-Haskell operational semantics . . . . .	26
2.15	ML-Scheme operational semantics . . . . .	27
2.16	Scheme grammar and evaluation contexts . . . . .	28
2.17	Scheme typing rules . . . . .	29
2.18	Scheme operational semantics . . . . .	30
2.19	Scheme-Haskell operational semantics . . . . .	31
2.20	Scheme-ML operational semantics . . . . .	32
2.21	Unbrand function . . . . .	33
2.22	Lump equality relation . . . . .	33

# Chapter 1

## Introduction

The complexities of software interoperation in part engender the proverbial reinvention of the wheel. Programmers forgo preexisting solutions to problems where interoperation proves too cumbersome; they reimplement software components, rather than reuse them. Disparate programming language features complicate the conversion of values exchanged between components of different languages. Resolving language incompatibilities transparently at boundaries between component languages facilitates interoperation by unburdening programmers. This paper explores and resolves two such incompatibilities with a model of computation and then proves its type soundness and describes its implementation.

The first incompatibility is type systems. Static type systems calculate and validate the types of expressions before run time, thereby ensuring that well-typed programs do not encounter type errors during run time. Dynamic type systems detect invalid operations on values using value predicates during run time and do not calculate or validate the types of expressions at compile time. Statically-typed languages—languages that use static type systems—that use values from



dynamically-typed languages must verify that the values match their expected types. Languages are assumed to exchange a common set of values that can be checked straightforwardly without coercion. Mismatched values and expected types could cause type errors during run time and violate type soundness. Ad-hoc polymorphism in dynamically-typed languages enables argument types to determine polymorphic function behavior. Since determining function behavior is undecidable [?], actual types for these functions cannot be reliably calculated at language boundaries and compared to expected types. Instead, they are wrapped in contracts [?] that defer the checking of their parameter and result types until they are used during run time. If they are never used, their types cannot be checked, but neither can they cause type errors.

The second incompatibility is parametricity. Parametric polymorphism in statically-typed languages enables function types to be abstracted with type variables and then instantiated into concrete types. Parametricity constrains the behavior of parametric polymorphic functions by ensuring that they behave the same regardless of the types and values of their arguments, and that functions with instantiated result types produce as their results the arguments associated with the same instantiated types. Functions from dynamically-typed languages that use value predicates or conditions on arguments and are used as parametric polymorphic functions by languages that have parametricity can violate their parametricity. Arguments for these functions must be obscured such that value predicates and conditions cannot examine them and annotated to ensure the correct ones are produced as results.

The third incompatibility is evaluation strategies. Evaluation strategies determine the order in which languages evaluate expressions. Eager evaluation evaluates expressions regardless of necessity, and lazy evaluation evaluates expressions

only where necessary. Lazy languages—languages that use lazy evaluation—can construct infinite streams as lists because they do not evaluate list elements when lists are constructed, but eager languages cannot because they do. Since there exist lazy lists—lists in lazy languages—for which no naturally equivalent eager lists exist, lazy lists crossing to eager languages are not converted to eager lists. Instead, elements of lazy lists are converted when accessed by eager languages if they are not lazy lists too.

The languages in the model must be able to express programs in which the aforementioned three incompatibilities arise. Haskell, ML, and Scheme each possess a unique combination of properties that together are sufficient for this purpose: Haskell and ML use static type systems and have parametricity, Scheme uses a dynamic type system, ML and Scheme use eager evaluation, and Haskell uses lazy evaluation.

The rest of the paper is organized as follows: Chapter 2 defines the model of computation, Chapter 3 proves the type soundness of the model, Chapter 4 describes an implementation of the model, Chapter 5 discusses related work, Chapter 6 discusses future work, and Chapter 7 discusses the conclusions.

# Chapter 2

## Model of Computation

The model of computation represents Haskell, ML, and Scheme with lambda calculus extended in various ways. Expressions represent software components, and nesting component expressions expresses interoperation between them, where the inner component expression evaluates to the value given to the outer component expression. Boundary expressions separate interoperating component expressions of different languages, indicate inner and outer languages, and declare the expected and actual types of the given values. The reduction of boundary expressions converts values between languages. The model extends the model of Kinghorn [?], which extended the model of Matthews and Findler [?].

The Haskell and ML models extend System F, which extends lambda calculus with explicit types that simplify the type soundness proofs and parametric polymorphism that approximates the type systems of Haskell and ML. The Scheme model extends lambda calculus with a simple type system to detect unbound variables.

Hereafter the names Haskell, ML, and Scheme refer to their corresponding

models, unless otherwise stated.

## 2.1 Grammars

Haskell types  $T$  comprise lumps  $L$ , natural numbers  $N$ , variables  $X$ , lists  $[T]$ , labels  $T^a$ , functions  $T \rightarrow T$ , and forall  $\forall X.T$ . Haskell values  $v_H$  comprise functions  $\lambda x : T.e_H$ , type abstractions  $\Lambda X.e_H$ , natural numbers  $\bar{n}$ , empty lists  $\text{nil}^T$ , list constructions  $\text{cons } e_H \ e_H$ , Scheme boundaries with lump expected types  ${}^LHS \ v_S$ , and Scheme boundaries with forall expected types  $\forall X.T \ HS \ v_S$ . Haskell expressions  $e_H$  comprise variables  $x$ , function applications  $e_H \ e_H$ , type applications  $e_H \ \{T\}$ , arithmetic operations  $+ \ e_H \ e_H$  and  $- \ e_H \ e_H$ , empty list predicates  $\text{null? } e_H$ , conditions  $\text{if0 } e_H \ e_H \ e_H$ , list operations  $\text{hd } e_H$  and  $\text{tl } e_H$ , fixed-point operations  $\text{fix } e_H$ , error reports  $\text{wrong}^T$  string, ML boundaries  ${}^THM^T \ e_M$ , and Scheme boundaries  ${}^THS \ e_S$ . Haskell evaluation contexts  $E_H$  conform to a call-by-name (lazy) evaluation strategy. Haskell holes are denoted  $[]_H$ . Figure 2.6 defines the Haskell grammar and evaluation contexts.

$\bar{n}$  syntactically represents the natural number  $n$ . The first subexpression in list constructions is the head and the second is the tail. Tails that are empty lists signify the ends of lists. Empty list predicates determine whether lists are empty. The first subexpression in conditions is the test, the second is the true alternative, and the third is the false alternative. Empty list predicates and conditions use the natural number zero as true and all other natural numbers as false. List operations produce the heads and tails of list constructions. Fixed-point operations render functions recursive. Error reports signal error conditions. ML and Scheme boundaries embed expressions from those languages in Haskell. Functions, empty lists, and error reports have type annotations that enable the

calculation of their types.

ML and Scheme have unforced values, which are forced values and Haskell boundaries, and forced values, which are

ML unforced values  $u_M$  comprise forced values  $v_M$  and Haskell boundaries  ${}^T M H^T e_H$ . ML forced values comprise functions, type abstractions, natural numbers, empty lists, list constructions  $\text{cons } v_M v_M$ , Scheme boundaries with lump expected types  ${}^L M S v_S$ , Scheme boundaries with forall expected types  ${}^{\forall X.T} M S v_S$ , and Haskell boundaries with list expected types  ${}^{[T]} M H^{[T]} (\text{cons } e_H e_H)$ . ML expressions comprise unforced values  $u_M$ , variables, function applications, type applications, arithmetic operations, empty list predicates, conditions, list constructions  $\text{cons } e_M e_M$ , list operations, fixed-point operations, error reports, and ML boundaries  ${}^T M S e_S$ . ML unforced evaluation contexts  $U_M$  do not force the reduction of Haskell boundaries and conform to an extended call-by-value (eager) evaluation strategy. ML forced evaluation contexts  $E_M$  force the reduction of Haskell boundaries. ML holes are denoted  $[]_M$ . Figure 2.11 defines the ML grammar and evaluation contexts.

Scheme unforced values comprise forced values and Haskell boundaries  $S H^T e_H$ . Scheme forced values comprise functions, natural numbers, empty lists, list constructions  $\text{cons } v_S v_S$ , Haskell boundaries with list actual types  $S H^{[T]} (\text{cons } e_H e_H)$ , Haskell boundaries with label actual types  $S H^{T^a} v_H$ , and ML boundaries with label actual types  $S M^{T^a} v_M$ . First, it does not have type abstractions, type applications, and fixed-point operations and the evaluation contexts that contain them. Second, it does not have types. Third, it does not have type annotations for functions, empty lists, and error reports. Fourth, it has three value predicate expressions that determine whether values are functions  $\text{fun? } e_S$ , lists  $\text{list? } e_S$ , and natural numbers  $\text{nat? } e_S$ . Figure 2.16 defines the Scheme grammar and

evaluation contexts.

Letter subscripts of grammar non-terminals denote the language to which they belong, and numbered superscripts denote individual instances of them. Variable and type variable names must be unique across all languages.

## 2.2 Typing Rules

Sch doesn't have Sys F stuff and , typing rules, and reduction rules that contain them

set membership

$\Gamma \vdash_H e_H : T$  denotes the Haskell typing relation. An expression  $e_H$  is well-typed within the context  $\Gamma$  if there is some type  $T$  such that  $\Gamma \vdash_H e_H : T$  is derivable.  $\Gamma \vdash_H T$  asserts the type  $T$  is well-formed within the context  $\Gamma$ . Where the context is empty, it is omitted from typing judgments. Programs that contain free variables or free type variables are ill-typed. Type equivalence is computed up to alpha-equivalence on bound type variables. Letter subscripts of type relations denote the language to which they belong.  $T_1[T_2/X]$  denotes the substitution of type  $T_2$  for free occurrences of type variable  $X$  within type  $T_1$ . Number subscripts and superscripts of grammar non-terminals in typing rules denote individual instances of them, but are absent where instances are unambiguous.

It has a single type is The Scheme Type,  $TST$ .

## 2.3 Operational Semantics

$e_H^1[e_H^2/x]$  denotes the substitution of expression  $e_H^2$  for free occurrences of variable  $x$  within expression  $e_H^1$ . Variable instances that occur on the right side of a reduction rule, but not its left, are new and unique. Error reports reduce to errors and terminate the computation. All reduction rules are defined with an unspecified evaluation context  $\mathcal{E}$ . The evaluation of a single language instantiates  $\mathcal{E}$  to  $E_H$  for Haskell, to  $E_M$  for ML, and to  $E_S$  for Scheme. Language interoperation instantiates  $\mathcal{E}$  according to the language in which programs begin and end.  $\mathcal{E}$  is implicitly instantiated correctly in later examples.

## 2.4 Interoperation Models

The interoperation calculi extend the core calculi with new expressions, evaluation contexts, typing rules, and reduction rules to enable interoperation. They add boundary expressions, which represent values with actual and expected types crossing between languages. Boundaries are denoted by two-letter acronyms, where the first letter names clients and the second letter names servers. Expected types are superscripts to the left of the first letters, and actual types are superscripts to the right of the second letters. Expressions to be reduced to values and cross languages are to the right of the letters and types, separated by a space. For example, the expression  $^{T_1}HM^{T_2} e_M$  denotes  $e_M$  with expected (Haskell) type  $T_1$  and actual (ML) type  $T_2$  crossing from ML to Haskell. Boundaries can be nested within each other to express interoperation between more than two languages. Since a set of  $n$  interoperable languages requires  $n \times (n - 1)$  boundaries, this model requires six boundaries.

They add evaluation contexts for the subexpressions of boundaries ( ${}^T H M^T E_M$  for example).

They add typing rules for boundaries. Boundaries are well-typed if their expected and actual types are well-formed and equivalent and the types of their subexpressions equal their actual types. The types of boundaries are their expected types.  $TST$  is omitted from boundary notation because all well-typed Scheme expressions have type  $TST$ .

They add reduction rules for every combination of boundary, expected and actual types, and syntactic forms of values. Rewrite rules for boundaries that contain Scheme values that do not match their expected types reduce to type error reports.

The expected and actual types of boundaries determine their reduction.

### 2.4.1 Natural Number Types

Natural numbers do not change when they are converted because the languages share the same number domain. For example,  ${}^N H M^N \bar{n}$  reduces to  $\bar{n}$ .

### 2.4.2 List Types

If a boundary has expected and actual list types ( ${}^{[T]} H M^{[T]} v_M$  for example), the value is either an empty list ( $\mathbf{nil}^T$  for example), a list construction ( $\mathbf{cons} v_M^1 v_M^2$  for example), or a Haskell list construction embedded in ML ( ${}^{[T]} M H^{[T]} (\mathbf{cons} e_H^1 e_H^2)$  for example). If it is an empty list, the boundary reduces to the empty list. For example,  ${}^{[T]} H M^{[T]} \mathbf{nil}^T$  reduces to  $\mathbf{nil}^T$ . If it is a list construction crossing from ML to Haskell, the boundary reduces to a list construction of the



old head and tail wrapped in boundaries. For example,  $^{[T]}HM^{[T]} (\mathbf{cons} \ v_M^1 \ v_M^2)$  reduces to  $\mathbf{cons} \ (^THM^T \ v_M^1) \ (^{[T]}HM^{[T]} \ v_M^2)$ . If it is a list construction crossing from Haskell to ML, the boundary is irreducible. Since Haskell list constructions can be infinite, they cannot be mechanically converted to equivalent ML list constructions. Therefore  $^{[T]}MH^{[T]} (\mathbf{cons} \ e_H^1 \ e_H^2)$  is a value. Instead, ML head and tail operations on embedded Haskell list constructions reduce to embedded heads and tails. For example,  $\mathbf{hd} \ (^{[T]}MH^{[T]} (\mathbf{cons} \ e_H^1 \ e_H^2))$  reduces to  $^THM^T \ e_H^1$ . If it is a Haskell list construction embedded in ML, the boundary reduces to the list construction. For example,  $^{[T]}HM^{[T]} (^{[T]}MH^{[T]} (\mathbf{cons} \ e_H^1 \ e_H^2))$  reduces to  $\mathbf{cons} \ e_H^1 \ e_H^2$ .

If a boundary has an expected Scheme type and an actual list type ( $SH^{[T]} \ v_H$  for example), the value is either an empty list, a list construction, or a Haskell list construction embedded in ML ( $^{[T]}MH^{[T]} (\mathbf{cons} \ e_H^1 \ e_H^2)$  for example). If it is an empty list, the boundary reduces to an empty list. For example,  $SH^{[T]} \mathbf{nil}^T$  reduces to  $\mathbf{nil}$ . If it is an ML list construction, the boundary reduces to a list construction of the old head and tail wrapped in boundaries. For example,  $SM^{[T]} (\mathbf{cons} \ v_M^1 \ v_M^2)$  reduces to  $\mathbf{cons} \ (SM^T \ v_M^1) \ (SM^{[T]} \ v_M^2)$ . If it is a Haskell list construction, it is irreducible for the same reason that  $^{[T]}MH^{[T]} (\mathbf{cons} \ e_H^1 \ e_H^2)$  is irreducible, as discussed above. Therefore  $SH^{[T]} (\mathbf{cons} \ e_H^1 \ e_H^2)$  is a value. Instead, Scheme head and tail operations on embedded Haskell list constructions reduce to embedded heads and tails. For example,  $\mathbf{hd} \ (SH^{[T]} (\mathbf{cons} \ e_H^1 \ e_H^2))$  reduces to  $SH^T \ e_H^1$ . If it is a Haskell list construction embedded in ML, the boundary reduces to the list construction embedded in Scheme. For example,  $SM^{[T]} (^{[T]}MH^{[T]} (\mathbf{cons} \ e_H^1 \ e_H^2))$  reduces to  $SH^{[T]} (\mathbf{cons} \ e_H^1 \ e_H^2)$ .

If a boundary has an expected list type and an actual Scheme type ( $^{[T]}HS \ v_S$  for example), the value is either an empty list, a list construction, or a Haskell

list construction embedded in Scheme ( $SH^{[T]} (\text{cons } e_H^1 e_H^2)$  for example). If it is an empty list, the boundary reduces to an empty list of the corresponding type. For example,  $^{[T]}HS \text{ nil}$  reduces to  $\text{nil}^T$ . If it is a list construction, the boundary reduces to a list construction of the old head and tail wrapped in boundaries. For example,  $^{[T]}HS (\text{cons } v_S^1 v_S^2)$  reduces to  $\text{cons } (^T HS v_S^1) (^{[T]} HS v_S^2)$ . If it is a Haskell list construction embedded in Scheme crossing to Haskell, the boundary reduces to the list construction. For example,  $^{[T]}HS (SH^{[T]} (\text{cons } e_H^1 e_H^2))$  reduces to  $\text{cons } e_H^1 e_H^2$ . If it is a Haskell list construction embedded in Scheme crossing to ML, the boundary reduces to the list construction embedded in ML. For example,  $^{[T]}MS (SH^{[T]} (\text{cons } e_H^1 e_H^2))$  reduces to  $^{[T]}MH^{[T]} (\text{cons } e_H^1 e_H^2)$ .

### 2.4.3 Function Types

Functions cannot be mechanically converted as they cross languages because the language grammars are different, Haskell and ML do not have a reasonable equivalent for every Scheme function, and functions may behave differently with different evaluation strategies. Instead, server functions are wrapped in client functions. The client functions apply the server function to their arguments and produce the results as their own. This is made possible by languages performing substitution within themselves across boundaries.

In Figure 2.1, a single boundary with an expected function type is split into two boundaries that convert the Haskell argument to an equivalent Scheme argument and the Scheme result to an equivalent Haskell result. Every boundary with a function type is split into two boundaries in this fashion. The Scheme-to-Haskell boundary verifies the syntactic form of its value matches its expected

$$\begin{aligned}
& ({}^{N \rightarrow [N]}HS \ \lambda x_1.\mathbf{nil}) \ \bar{0} \\
\rightarrow & \ (\lambda x_2 : N.({}^{[N]}HS \ ((\lambda x_1.\mathbf{nil}) \ (SH^N \ x_2)))) \ \bar{0} \\
\rightarrow & \ {}^{[N]}HS \ ((\lambda x_1.\mathbf{nil}) \ (SH^N \ \bar{0})) \\
\rightarrow & \ {}^{[N]}HS \ ((\lambda x_1.\mathbf{nil}) \ \bar{0}) \\
\rightarrow & \ {}^{[N]}HS \ \mathbf{nil} \\
\rightarrow & \ \mathbf{nil}^N
\end{aligned}$$

**Figure 2.1: Conversion of a function**

type. If its value is not some list of natural numbers, it reports a type error. If the body of the Scheme function had been  $\bar{0}$  instead of  $\mathbf{nil}$ , the computation would have reduced to  ${}^{[N]}HS \ \bar{0}$  instead of  ${}^{[N]}HS \ \mathbf{nil}$ . Since  $\bar{0}$  is not a list,  ${}^{[N]}HS \ \bar{0}$  reduces to **wrong** “Not a list” to report the type error.

The case for higher-order functions is more complex, but straightforward. See Figure 2.2 for an example.

#### 2.4.4 Forall Types

If a boundary has expected and actual forall types ( $\forall^{X.T}HM^{\forall^{X.T}} v_M$  for example), the value is either a type abstraction ( $\Lambda X.e_M$  for example) or a Scheme value wrapped in an inner boundary ( $\forall^{X.T}MS v_S$  for example). If it is a type abstraction, the boundary moves inside the type abstraction and wraps the expression. For example,  $\forall^{X.T}HM^{\forall^{X.T}} \Lambda X.e_M$  reduces to  $\Lambda X.({}^THM^T e_M)$ . If it is a Scheme value wrapped in an inner boundary, the outer boundary reduces to a new boundary bridging the outer language and Scheme that contains the Scheme value. For example,  $\forall^{X.T}HM^{\forall^{X.T}} (\forall^{X.T}MS v_S)$  reduces to  $\forall^{X.T}HS v_S$ .

If a boundary has an expected forall type and an actual Scheme type ( $\forall^{X.T}HS$

$$\begin{aligned}
& ((^{N \rightarrow N} \rightarrow^N HS \ \lambda x_1.(x_1 \ \bar{0})) \ (\lambda x_2 : N.x_2)) \\
\rightarrow & \ (\lambda x_3 : N \rightarrow N. (^N HS \ ((\lambda x_1.(x_1 \ \bar{0})) \ (SH^{N \rightarrow N} \ x_3)))) \ (\lambda x_2 : N.x_2) \\
\rightarrow & \ ^N HS \ ((\lambda x_1.(x_1 \ \bar{0})) \ (SH^{N \rightarrow N} \ (\lambda x_2 : N.x_2))) \\
\rightarrow & \ ^N HS \ ((\lambda x_1.(x_1 \ \bar{0})) \ (\lambda x_4.(SH^N \ ((\lambda x_2 : N.x_2) \ (^N HS \ x_4))))) \\
\rightarrow & \ ^N HS \ ((\lambda x_4.(SH^N \ ((\lambda x_2 : N.x_2) \ (^N HS \ x_4)))) \ \bar{0}) \\
\rightarrow & \ ^N HS \ (SH^N \ ((\lambda x_2 : N.x_2) \ (^N HS \ \bar{0}))) \\
\rightarrow & \ ^N HS \ (SH^N \ ((\lambda x_2 : N.x_2) \ \bar{0})) \\
\rightarrow & \ ^N HS \ (SH^N \ \bar{0}) \\
\rightarrow & \ ^N HS \ \bar{0} \\
\rightarrow & \ \bar{0}
\end{aligned}$$

**Figure 2.2: Conversion of a higher-order function**

$v_S$  for example), the value is a Scheme value. Such a boundary is irreducible because Scheme does not have type abstractions. Therefore  $\forall^{X.T} HS \ v_S$  and  $\forall^{X.T} MS \ v_S$  are values. Nevertheless, there are useful Scheme values that correspond to forall types, and they ought to be convertible. If the expected forall type is instantiated and the result is not a forall type, the boundary is reducible and the Scheme value is convertible. However, Haskell and ML preserve parametricity, and instantiating the expected forall type does nothing to prevent the Scheme value, if it is a function, from breaking parametricity once converted.

Scheme functions with expected forall types can break parametricity by using value predicates and conditions to determine their behavior by the types and values of their arguments. Haskell and ML must wrap their arguments for these functions such that Scheme value predicates and conditions cannot examine them. Expected forall types of boundaries can be instantiated by applying those boundaries to types. These type applications label their type arguments with

$$\begin{aligned}
& ((\forall X.(X \rightarrow X) HS \lambda x_1.(\text{if0 } x_1 \bar{1} x_1)) \{N\}) \bar{0} \\
\rightarrow & \quad ({}^{N^a \rightarrow N^a} HS \lambda x_1.(\text{if0 } x_1 \bar{1} x_1)) \bar{0} \\
\rightarrow & \quad (\lambda x_2 : N.({}^{N^a} HS ((\lambda x_1.(\text{if0 } x_1 \bar{1} x_1)) (SH^{N^a} x_2)))) \bar{0} \\
\rightarrow & \quad {}^{N^a} HS ((\lambda x_1.(\text{if0 } x_1 \bar{1} x_1)) (SH^{N^a} \bar{0})) \\
\rightarrow & \quad {}^{N^a} HS (\text{if0 } (SH^{N^a} \bar{0}) \bar{1} (SH^{N^a} \bar{0})) \\
\rightarrow & \quad {}^{N^a} HS (SH^{N^a} \bar{0}) \\
\rightarrow & \quad \bar{0}
\end{aligned}$$

**Figure 2.3: Labels protect parametricity**

unique labels, denoted  $T^a$ , before instantiating the expected forall types with them. Boundaries with actual label types ( $SH^{T^a} e_H$  for example) are irreducible; Scheme value predicates and conditions cannot examine them. Therefore  $SH^{T^a} e_H$  and  $SM^{T^a} v_M$  are values. Scheme can return these wrapped arguments to Haskell and ML if the expected and actual types match. For example,  ${}^{T^a} HS (SH^{T^a} e_H)$  reduces to  $e_H$ . If they do not match, the outer boundary reduces to a parametricity error report. See Figure 2.3 for an example.

Since the Haskell and ML typing relations expect type applications to substitute types unchanged, they expect  $\forall X.(X \rightarrow X)$ , instantiated with  $N$ , to be  $N \rightarrow N$ . Observe that the application of  $\forall X.(X \rightarrow X) HS \lambda x_1.(\text{if0 } x_1 \bar{1} x_1)$ , which has type  $\forall X.(X \rightarrow X)$ , to  $N$  reduces to  ${}^{N^a \rightarrow N^a} HS \lambda x_1.(\text{if0 } x_1 \bar{1} x_1)$ , which appears to have type  $N^a \rightarrow N^a$ . The Haskell and ML typing relations resolve this conflict by removing all labels from expected and actual types before making typing judgements. Therefore  ${}^{N^a \rightarrow N^a} HS \lambda x_1.(\text{if0 } x_1 \bar{1} x_1)$  has type  $N \rightarrow N$ , as expected. Rewrite rules remove labels where required to resolve type conflicts.  $T[T_i/T_i^a]$  denotes the replacement of every label type  $T_i^a$  with its underlying type  $T_i$  within  $T$ .

$$\begin{aligned}
& (((\forall X_1.(\forall X_2.(X_1 \rightarrow (X_2 \rightarrow X_2))) HS \lambda x_1.(\lambda x_2.x_1)) \{N\}) \{N\}) \bar{0}) \bar{1} \\
\rightarrow & (((\forall X_2.(N^a \rightarrow (X_2 \rightarrow X_2)) HS \lambda x_1.(\lambda x_2.x_1)) \{N\}) \bar{0}) \bar{1} \\
\rightarrow & ((N^a \rightarrow (N^b \rightarrow N^b) HS \lambda x_1.(\lambda x_2.x_1)) \bar{0}) \bar{1} \\
\rightarrow & ((\lambda x_3 : N.(N^b \rightarrow N^b HS ((\lambda x_1.(\lambda x_2.x_1)) (SH^{N^a} x_3)))) \bar{0}) \bar{1} \\
\rightarrow & (N^b \rightarrow N^b HS ((\lambda x_1.(\lambda x_2.x_1)) (SH^{N^a} \bar{0}))) \bar{1} \\
\rightarrow & (\lambda x_4 : N.(N^b HS (((\lambda x_1.(\lambda x_2.x_1)) (SH^{N^a} \bar{0})) (SH^{N^b} x_4)))) \bar{1} \\
\rightarrow & N^b HS (((\lambda x_1.(\lambda x_2.x_1)) (SH^{N^a} \bar{0})) (SH^{N^b} \bar{1})) \\
\rightarrow & N^b HS ((\lambda x_2.(SH^{N^a} \bar{0})) (SH^{N^b} \bar{1})) \\
\rightarrow & N^b HS (SH^{N^a} \bar{0}) \\
\rightarrow & \text{wrong "Parametricity violated"} \\
\rightarrow & \text{Error: "Parametricity violated"}
\end{aligned}$$

**Figure 2.4: Labels detect parametricity violations**

Scheme functions with expected forall types can also break parametricity by producing the wrong argument as their results. Haskell and ML assume type variables for result types are instantiated along with one or more type variables for argument types. For example, Haskell and ML assume a function with type  $\forall X_1.(\forall X_2.(X_1 \rightarrow (X_2 \rightarrow X_2)))$  reduces to its second argument because the second argument and the result share the same type variable. Labels enable Haskell and ML to detect and report violations of these assumptions during run time. Since unique labels are used for each application of a boundary to a type, they group argument and result types together. Mismatched labels for expected and actual types of boundaries indicates that Scheme broke parametricity. See Figure 2.4 for an example.

If a boundary has an expected Scheme type and an actual forall type ( $SH^{\forall X.T} e_H$  for example), the value is either a type abstraction ( $\Lambda X.e_H$  for example)

or a Scheme value wrapped in an inner boundary ( $\forall^{X.T} HS \ v_S$  for example). If the value is a type abstraction, the boundary is irreducible because Scheme does not have type abstractions. Instead, the actual type is instantiated with, and the type abstraction is applied to, the lump type, denoted  $L$ . If the result is not another type abstraction, the boundary is reducible. Boundaries with expected lump types are irreducible;  ${}^L HS \ v_S$  and  ${}^L MS \ v_S$  are values. Empty lists instantiated with the lump type convert as with other types because their conversions discard their type annotations. Likewise, error reports instantiated with the lump type terminate the computation as with other types. Polymorphic functions instantiated with the lump type satisfy the expectations of all languages because they convert to Scheme functions that can be applied to arguments of various types, but do not break parametricity. These polymorphic functions can return their arguments to Scheme if the expected and actual types are lump types. For example,  $SH^L ({}^L HS \ v_S)$  reduces to  $v_S$ . See Figure 2.5 for an example.

If it is a Scheme value wrapped in an inner boundary, the outer boundary reduces to the Scheme value. For example,  $SH^{\forall^{X.T}} (\forall^{X.T} HS \ v_S)$  reduces to  $v_S$ .

$$\begin{aligned}
& (\lambda x_1.(\mathbf{cons} (x_1 \ \bar{0}) (x_1 \ \mathbf{nil}))) (SH^{\forall X.(X \rightarrow X)} \ \Lambda X.(\lambda x_2 : X.x_2)) \\
\rightarrow & (\lambda x_1.(\mathbf{cons} (x_1 \ \bar{0}) (x_1 \ \mathbf{nil}))) (SH^{L \rightarrow L} ((\Lambda X.(\lambda x_2 : X.x_2)) \ \{L\})) \\
\rightarrow & (\lambda x_1.(\mathbf{cons} (x_1 \ \bar{0}) (x_1 \ \mathbf{nil}))) (SH^{L \rightarrow L} \ \lambda x_2 : L.x_2) \\
\rightarrow & (\lambda x_1.(\mathbf{cons} (x_1 \ \bar{0}) (x_1 \ \mathbf{nil}))) (\lambda x_3.(SH^L ((\lambda x_2 : L.x_2) (^LHS \ x_3)))) \\
\rightarrow & \mathbf{cons} ((\lambda x_3.(SH^L ((\lambda x_2 : L.x_2) (^LHS \ x_3)))) \ \bar{0}) \\
& ((\lambda x_3.(SH^L ((\lambda x_2 : L.x_2) (^LHS \ x_3)))) \ \mathbf{nil})) \\
\rightarrow & \mathbf{cons} (SH^L ((\lambda x_2 : L.x_2) (^LHS \ \bar{0}))) \\
& ((\lambda x_3.(SH^L ((\lambda x_2 : L.x_2) (^LHS \ x_3)))) \ \mathbf{nil})) \\
\rightarrow & \mathbf{cons} (SH^L (^LHS \ \bar{0})) ((\lambda x_3.(SH^L ((\lambda x_2 : L.x_2) (^LHS \ x_3)))) \ \mathbf{nil})) \\
\rightarrow & \mathbf{cons} \ \bar{0} ((\lambda x_3.(SH^L ((\lambda x_2 : L.x_2) (^LHS \ x_3)))) \ \mathbf{nil})) \\
\rightarrow & \mathbf{cons} \ \bar{0} (SH^L ((\lambda x_2 : L.x_2) (^LHS \ \mathbf{nil}))) \\
\rightarrow & \mathbf{cons} \ \bar{0} (SH^L (^LHS \ \mathbf{nil})) \\
\rightarrow & \mathbf{cons} \ \bar{0} \ \mathbf{nil}
\end{aligned}$$

**Figure 2.5: Polymorphic function converted to Scheme function**



$$\begin{aligned}
e_H &= x_H \mid v_H \mid e_H e_H \mid e_H \langle t_H \rangle \mid \mathbf{fix} \, e_H \mid o \, e_H e_H \mid \mathbf{if0} \, e_H e_H e_H \mid f \, e_H \\
&\quad \mathbf{null?} \, e_H \mid \mathbf{wrong} \, t_H \, string \mid \mathbf{hm} \, t_H \, t_M \, e_M \mid \mathbf{hs} \, k_H \, e_S \\
v_H &= \lambda x_H : t_H . e_H \mid \Lambda u_H . e_H \mid \bar{n} \mid \mathbf{nil} \, t_H \mid \mathbf{cons} \, e_H \, e_H \mid \mathbf{hm} \, L \, t_M \, w_M \\
&\quad \mathbf{hs} \, L \, w_S \\
t_H &= L \mid N \mid u_H \mid \{t_H\} \mid t_H \rightarrow t_H \mid \forall u_H . t_H \\
k_H &= L \mid N \mid u_H \mid \{k_H\} \mid k_H \rightarrow k_H \mid \forall u_H . k_H \mid b \diamond t_H \\
o &= + \mid - \\
f &= \mathbf{hd} \mid \mathbf{tl} \\
E_H &= []_H \mid E_H e_H \mid E_H \langle t_H \rangle \mid \mathbf{fix} \, E_H \mid o \, E_H e_H \mid o \, v_H \, E_H \\
&\quad \mathbf{if0} \, E_H e_H e_H \mid f \, E_H \mid \mathbf{null?} \, E_H \mid \mathbf{hm} \, t_H \, t_M \, E_M \mid \mathbf{hs} \, k_H \, E_S
\end{aligned}$$

**Figure 2.6: Haskell grammar and evaluation contexts**

$$\begin{array}{c}
\overline{\vdash_H \mathbf{L}} \quad \overline{\vdash_H \mathbf{N}} \quad \overline{\Gamma, u_H \vdash_H u_H} \\
\frac{\Gamma \vdash_H t_H}{\Gamma \vdash_H \{t_H\}} \quad \frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_H t'_H}{\Gamma \vdash_H t_H \rightarrow t'_H} \quad \frac{\Gamma, u_H \vdash_H t_H}{\Gamma \vdash_H \forall u_H. t_H} \\
\\
\frac{\Gamma \vdash_H t_H \quad \Gamma, x_H : t_H \vdash_H e_H : t'_H}{\Gamma \vdash_H (\lambda x_H : t_H. e_H) : t_H \rightarrow t'_H} \quad \frac{\Gamma, u_H \vdash_H e_H : t_H}{\Gamma \vdash_H \Lambda u_H. e_H : \forall u_H. t_H} \quad \overline{\vdash_H \bar{n} : \mathbf{N}} \\
\\
\frac{\Gamma \vdash_H t_H : \{t_H\}}{\Gamma \vdash_H \mathbf{nil} \, t_H : \{t_H\}} \quad \frac{\Gamma \vdash_H e_H : t_H \quad \Gamma \vdash_H e'_H : \{t_H\}}{\Gamma \vdash_H \mathbf{cons} \, e_H \, e'_H : \{t_H\}} \quad \overline{\Gamma, x_H : t_H \vdash_H x_H : t_H} \\
\\
\frac{\Gamma \vdash_H e_H : t_H \rightarrow t'_H \quad \Gamma \vdash_H e'_H : t_H}{\Gamma \vdash_H e_H \, e'_H : t'_H} \quad \frac{\Gamma \vdash_H e_H : t_H \rightarrow t_H}{\Gamma \vdash_H \mathbf{fix} \, e_H : t_H} \\
\\
\frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_H e_H : \forall u_H. t'_H}{\Gamma \vdash_H e_H \langle t_H \rangle : t'_H[t_H/u_H]} \quad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathbf{hd} \, e_H : t_H} \quad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathbf{tl} \, e_H : \{t_H\}} \\
\\
\frac{\Gamma \vdash_H e_H : \mathbf{N} \quad \Gamma \vdash_H e'_H : \mathbf{N}}{\Gamma \vdash_H o \, e_H \, e'_H : \mathbf{N}} \quad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathbf{null?} \, e_H : \mathbf{N}} \quad \frac{\Gamma \vdash_H [k_H] \quad \Gamma \vdash_S e_S : \mathbf{TST}}{\Gamma \vdash_H \mathbf{hs} \, k_H \, e_S : [k_H]} \\
\\
\frac{\Gamma \vdash_H e_H : \mathbf{N} \quad \Gamma \vdash_H e'_H : t_H \quad \Gamma \vdash_H e''_H : t_H}{\Gamma \vdash_H \mathbf{if0} \, e_H \, e'_H \, e''_H : t_H} \quad \frac{\Gamma \vdash_H t_H}{\Gamma \vdash_H \mathbf{wrong} \, t_H \, \mathit{string} : t_H} \\
\\
\frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_M t_M \quad \Gamma \vdash_M e_M : t'_M \quad t_H \doteq t_M \quad t_M = t'_M}{\Gamma \vdash_H \mathbf{hm} \, t_H \, t_M \, e_M : t_H}
\end{array}$$

**Figure 2.7: Haskell typing rules**

$$\begin{aligned}
& \mathcal{E}[(\lambda x_H : t_H.e_H) e'_H]_H \rightarrow \mathcal{E}[e_H[e'_H/x_H]] \\
& \mathcal{E}[(\Lambda u_H.e_H)\langle t_H \rangle]_H \rightarrow \mathcal{E}[e_H[b \diamond t_H/u_H]] \\
& \mathcal{E}[\mathbf{fix} (\lambda x_H : t_H.e_H)]_H \rightarrow \mathcal{E}[e_H[\mathbf{fix} (\lambda x_H : t_H.e_H)/x_H]] \\
& \mathcal{E}[+ \bar{n} \bar{n'}]_H \rightarrow \mathcal{E}[\overline{n + n'}] \\
& \mathcal{E}[- \bar{n} \bar{n'}]_H \rightarrow \mathcal{E}[\overline{\max(n - n', 0)}] \\
& \mathcal{E}[\mathbf{if0} \bar{0} e_H e'_H]_H \rightarrow \mathcal{E}[e_H] \\
& \mathcal{E}[\mathbf{if0} \bar{n} e_H e'_H]_H \rightarrow \mathcal{E}[e'_H] \ (n \neq 0) \\
& \mathcal{E}[\mathbf{hd} (\mathbf{nil} t_H)]_H \rightarrow \mathcal{E}[\mathbf{wrong} t_H \text{ “Empty list”}] \\
& \mathcal{E}[\mathbf{tl} (\mathbf{nil} t_H)]_H \rightarrow \mathcal{E}[\mathbf{wrong} \{t_H\} \text{ “Empty list”}] \\
& \mathcal{E}[\mathbf{hd} (\mathbf{cons} e_H e'_H)]_H \rightarrow \mathcal{E}[e_H] \\
& \mathcal{E}[\mathbf{tl} (\mathbf{cons} e_H e'_H)]_H \rightarrow \mathcal{E}[e'_H] \\
& \mathcal{E}[\mathbf{null?} (\mathbf{nil} t_H)]_H \rightarrow \mathcal{E}[\bar{0}] \\
& \mathcal{E}[\mathbf{null?} (\mathbf{cons} e_H e'_H)]_H \rightarrow \mathcal{E}[\bar{1}] \\
& \mathcal{E}[\mathbf{wrong} t_H \text{ string}]_H \rightarrow \mathbf{Error: string}
\end{aligned}$$

**Figure 2.8: Haskell operational semantics**

$$\begin{aligned}
& \mathcal{E}[\mathbf{hm} \ t_H \ \mathbf{L} \ (\mathbf{mh} \ \mathbf{L} \ t'_H \ e_H)]_H \rightarrow \mathcal{E}[e_H] \quad (t_H = t'_H \text{ and } t_H \neq \mathbf{L}) \\
& \mathcal{E}[\mathbf{hm} \ t_H \ \mathbf{L} \ (\mathbf{mh} \ \mathbf{L} \ t'_H \ e_H)]_H \rightarrow \mathcal{E}[\mathbf{wrong} \ t_H \ \text{“Type mismatch”}] \\
& \quad (t_H \neq t'_H \text{ and } t_H \neq \mathbf{L}) \\
& \mathcal{E}[\mathbf{hm} \ t_H \ \mathbf{L} \ (\mathbf{ms} \ \mathbf{L} \ w_S)]_H \rightarrow \mathcal{E}[\mathbf{wrong} \ t_H \ \text{“Bad value”}] \quad (t_H \neq \mathbf{L}) \\
& \mathcal{E}[\mathbf{hm} \ \mathbf{N} \ \mathbf{N} \ \bar{n}]_H \rightarrow \mathcal{E}[\bar{n}] \\
& \mathcal{E}[\mathbf{hm} \ \{t_H\} \ \{t_M\} \ (\mathbf{nil} \ t'_M)]_H \rightarrow \mathcal{E}[\mathbf{nil} \ t_H] \\
& \mathcal{E}[\mathbf{hm} \ \{t_H\} \ \{t_M\} \ (\mathbf{cons} \ v_M \ v'_M)]_H \rightarrow \\
& \quad \mathcal{E}[\mathbf{cons} \ (\mathbf{hm} \ t_H \ t_M \ v_M) \ (\mathbf{hm} \ \{t_H\} \ \{t_M\} \ v'_M)] \\
& \mathcal{E}[\mathbf{hm} \ (t_H \rightarrow t'_H) \ (t_M \rightarrow t'_M) \ (\lambda x_M : t''_M . e_M)]_H \rightarrow \\
& \quad \mathcal{E}[\lambda x_H : t_H . \mathbf{hm} \ t'_H \ t'_M \ ((\lambda x_M : t''_M . e_M) \ (\mathbf{mh} \ t_M \ t_H \ x_H))] \\
& \mathcal{E}[\mathbf{hm} \ (\forall u_H . t_H) \ (\forall u_M . t_M) \ (\Lambda u'_M . e_M)]_H \rightarrow \mathcal{E}[\Lambda u_H . \mathbf{hm} \ t_H \ t_M [\mathbf{L}/u_M] \ e_M [\mathbf{L}/u'_M]]
\end{aligned}$$

**Figure 2.9: Haskell-ML operational semantics**

$$\begin{aligned}
\mathcal{E}[\mathbf{hs} \ N \ \bar{n}]_H &\rightarrow \mathcal{E}[\bar{n}] \\
\mathcal{E}[\mathbf{hs} \ N \ w_S]_H &\rightarrow \mathcal{E}[\mathbf{wrong} \ N \ \text{“Not a number”}] \ (w_S \neq \bar{n}) \\
\mathcal{E}[\mathbf{hs} \ \{k_H\} \ \mathbf{nil}]_H &\rightarrow \mathcal{E}[\mathbf{nil} \ \lfloor k_H \rfloor] \\
\mathcal{E}[\mathbf{hs} \ \{k_H\} \ (\mathbf{cons} \ v_S \ v'_S)]_H &\rightarrow \mathcal{E}[\mathbf{cons} \ (\mathbf{hs} \ k_H \ v_S) \ (\mathbf{hs} \ \{k_H\} \ v'_S)] \\
\mathcal{E}[\mathbf{hs} \ \{k_H\} \ w_S]_H &\rightarrow \mathcal{E}[\mathbf{wrong} \ \lfloor \{k_H\} \rfloor \ \text{“Not a list”}] \\
&\quad (w_S \neq \mathbf{nil} \text{ and } w_S \neq \mathbf{cons} \ v_S \ v'_S) \\
\mathcal{E}[\mathbf{hs} \ (b \diamond t_H) \ (\mathbf{sh} \ (b \diamond t_H) \ e_H)]_H &\rightarrow \mathcal{E}[e_H] \\
\mathcal{E}[\mathbf{hs} \ (b \diamond t_H) \ w_S]_H &\rightarrow \mathcal{E}[\mathbf{wrong} \ t_H \ \text{“Brand mismatch”}] \ (w_S \neq \mathbf{sh} \ (b \diamond t_H) \ e_H) \\
\mathcal{E}[\mathbf{hs} \ (k_H \rightarrow k'_H) \ (\lambda x_S. e_S)]_H &\rightarrow \mathcal{E}[\lambda x_H : \lfloor k_H \rfloor. \mathbf{hs} \ k'_H \ ((\lambda x_S. e_S) \ (\mathbf{sh} \ k_H \ x_H))] \\
\mathcal{E}[\mathbf{hs} \ (k_H \rightarrow k'_H) \ w_S]_H &\rightarrow \mathcal{E}[\mathbf{wrong} \ \lfloor k_H \rightarrow k'_H \rfloor \ \text{“Not a function”}] \\
&\quad (w_S \neq \lambda x_S. e_S) \\
\mathcal{E}[\mathbf{hs} \ (\forall u_H. k_H) \ w_S]_H &\rightarrow \mathcal{E}[\Lambda u_H. \mathbf{hs} \ k_H \ w_S]
\end{aligned}$$

**Figure 2.10: Haskell-Scheme operational semantics**

$$\begin{aligned}
e_M &= x_M \mid v_M \mid e_M e_M \mid e_M \langle t_M \rangle \mid \mathbf{fix} \ e_M \mid o \ e_M \ e_M \mid \mathbf{if0} \ e_M \ e_M \ e_M \\
&\quad \mathbf{cons} \ e_M \ e_M \mid f \ e_M \mid \mathbf{null?} \ e_M \mid \mathbf{wrong} \ t_M \ \mathit{string} \mid \mathbf{ms} \ k_M \ e_S \\
v_M &= w_M \mid \mathbf{mh} \ t_M \ t_H \ e_H \\
w_M &= \lambda x_M : t_M . e_M \mid \Lambda u_M . e_M \mid \bar{n} \mid \mathbf{nil} \ t_M \mid \mathbf{cons} \ v_M \ v_M \mid \mathbf{mh} \ L \ t_H \ e_H \\
&\quad \mathbf{ms} \ L \ w_S \\
t_M &= L \mid N \mid u_M \mid \{t_M\} \mid t_M \rightarrow t_M \mid \forall u_M . t_M \\
k_M &= L \mid N \mid u_M \mid \{k_M\} \mid k_M \rightarrow k_M \mid \forall u_M . k_M \mid b \diamond t_M \\
o &= + \mid - \\
f &= \mathbf{hd} \mid \mathbf{tl} \\
E_M &= U_M \mid \mathbf{mh} \ t_M \ t_H \ E_H \\
U_M &= []_M \mid E_M \ e_M \mid w_M \ U_M \mid E_M \langle t_M \rangle \mid \mathbf{fix} \ E_M \mid o \ E_M \ e_M \mid o \ w_M \ E_M \\
&\quad \mathbf{if0} \ E_M \ e_M \ e_M \mid \mathbf{cons} \ U_M \ e_M \mid \mathbf{cons} \ v_M \ U_M \mid f \ E_M \mid \mathbf{null?} \ E_M \\
&\quad \mathbf{ms} \ k_M \ E_S
\end{aligned}$$

**Figure 2.11:** ML grammar and evaluation contexts

$$\begin{array}{c}
\overline{\vdash_M \mathbf{L}} \quad \overline{\vdash_M \mathbf{N}} \quad \overline{\Gamma, u_M \vdash_M u_M} \\
\frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \{t_M\}} \quad \frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_M t'_M}{\Gamma \vdash_M t_M \rightarrow t'_M} \quad \frac{\Gamma, u_M \vdash_M t_M}{\Gamma \vdash_M \forall u_M. t_M} \\
\\
\frac{\Gamma \vdash_M t_M \quad \Gamma, x_M : t_M \vdash_M e_M : t'_M}{\Gamma \vdash_M (\lambda x_M : t_M. e_M) : t_M \rightarrow t'_M} \quad \frac{\Gamma, u_M \vdash_M e_M : t_M}{\Gamma \vdash_M \Lambda u_M. e_M : \forall u_M. t_M} \quad \overline{\vdash_M \bar{n} : \mathbf{N}} \\
\\
\frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \mathbf{nil} \ t_M : \{t_M\}} \quad \frac{\Gamma \vdash_M e_M : t_M \quad \Gamma \vdash_M e'_M : \{t_M\}}{\Gamma \vdash_M \mathbf{cons} \ e_M \ e'_M : \{t_M\}} \quad \frac{}{\Gamma, x_M : t_M \vdash_M x_M : t_M} \\
\\
\frac{\Gamma \vdash_M e_M : t_M \rightarrow t'_M \quad \Gamma \vdash_M e'_M : t_M}{\Gamma \vdash_H e_M \ e'_M : t'_M} \quad \frac{\Gamma \vdash_M e_M : t_M \rightarrow t_M}{\Gamma \vdash_M \mathbf{fix} \ e_M : t_M} \\
\\
\frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_M e_M : \forall u_M. t'_M}{\Gamma \vdash_M e_M \langle t_M \rangle : t'_M[t_M/u_M]} \quad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathbf{hd} \ e_M : t_M} \quad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathbf{tl} \ e_M : \{t_M\}} \\
\\
\frac{\Gamma \vdash_M e_M : \mathbf{N} \quad \Gamma \vdash_M e'_M : \mathbf{N}}{\Gamma \vdash_M o \ e_M \ e'_M : \mathbf{N}} \quad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathbf{null?} \ e_M : \mathbf{N}} \quad \frac{\Gamma \vdash_M [k_M] \quad \Gamma \vdash_S e_S : \mathbf{TST}}{\Gamma \vdash_M \mathbf{ms} \ k_M \ e_S : [k_M]} \\
\\
\frac{\Gamma \vdash_M e_M : \mathbf{N} \quad \Gamma \vdash_M e'_M : t_M \quad \Gamma \vdash_M e''_M : t_M}{\Gamma \vdash_M \mathbf{if0} \ e_M \ e'_M \ e''_M : t_M} \quad \frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \mathbf{wrong} \ t_M \ \mathit{string} : t_M} \\
\\
\frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_H t_H \quad \Gamma \vdash_H e_H : t'_H \quad t_M \doteq t_H \quad t_H = t'_H}{\Gamma \vdash_M \mathbf{mh} \ t_M \ t_H \ e_H : t_M}
\end{array}$$

**Figure 2.12:** ML typing rules

$$\begin{aligned}
& \mathcal{E}[(\lambda x_M : t_M.e_M) v_M]_M \rightarrow \mathcal{E}[e_M[v_M/x_M]] \\
& \mathcal{E}[(\Lambda u_M.e_M)\langle t_M \rangle]_M \rightarrow \mathcal{E}[e_M[b \diamond t_M/u_M]] \\
& \mathcal{E}[\mathbf{fix} (\lambda x_M : t_M.e_M)]_M \rightarrow \mathcal{E}[e_M[\mathbf{fix} (\lambda x_M : t_M.e_M)/x_M]] \\
& \mathcal{E}[+ \bar{n} \bar{n'}]_M \rightarrow \mathcal{E}[\overline{n + n'}] \\
& \mathcal{E}[- \bar{n} \bar{n'}]_M \rightarrow \mathcal{E}[\overline{\max(n - n', 0)}] \\
& \mathcal{E}[\mathbf{if0} \bar{0} e_M e'_M]_M \rightarrow \mathcal{E}[e_M] \\
& \mathcal{E}[\mathbf{if0} \bar{n} e_M e'_M]_M \rightarrow \mathcal{E}[e'_M] \ (n \neq 0) \\
& \mathcal{E}[\mathbf{hd} (\mathbf{nil} t_M)]_M \rightarrow \mathcal{E}[\mathbf{wrong} t_M \text{ “Empty list”}] \\
& \mathcal{E}[\mathbf{tl} (\mathbf{nil} t_M)]_M \rightarrow \mathcal{E}[\mathbf{wrong} \{t_M\} \text{ “Empty list”}] \\
& \mathcal{E}[\mathbf{hd} (\mathbf{cons} v_M v'_M)]_M \rightarrow \mathcal{E}[v_M] \\
& \mathcal{E}[\mathbf{tl} (\mathbf{cons} v_M v'_M)]_M \rightarrow \mathcal{E}[v'_M] \\
& \mathcal{E}[\mathbf{null?} (\mathbf{nil} t_M)]_M \rightarrow \mathcal{E}[\bar{0}] \\
& \mathcal{E}[\mathbf{null?} (\mathbf{cons} v_M v'_M)]_M \rightarrow \mathcal{E}[\bar{1}] \\
& \mathcal{E}[\mathbf{wrong} t_M \text{ string}]_H \rightarrow \mathbf{Error: string}
\end{aligned}$$

**Figure 2.13:** ML operational semantics



$$\begin{aligned}
& \mathcal{E}[\text{mh } t_M \text{ L } (\text{hm L } t'_M w_M)]_M \rightarrow \mathcal{E}[w_M] \text{ } (t_M = t'_M \text{ and } t_M \neq \text{L}) \\
& \mathcal{E}[\text{mh } t_M \text{ L } (\text{hm L } t'_M w_M)]_M \rightarrow \mathcal{E}[\text{wrong } t_M \text{ "Type mismatch"}] \text{ } (t_M \neq t'_M \text{ and } t_M \neq \text{L}) \\
& \mathcal{E}[\text{mh } t_M \text{ L } (\text{hs L } w_S)]_H \rightarrow \mathcal{E}[\text{wrong } t_M \text{ "Bad value"}] \text{ } (t_M \neq \text{L}) \\
& \mathcal{E}[\text{mh N N } \bar{n}]_M \rightarrow \mathcal{E}[\bar{n}] \\
& \mathcal{E}[\text{mh } \{t_M\} \{t_H\} (\text{nil } t'_H)]_M \rightarrow \mathcal{E}[\text{nil } t_M] \\
& \mathcal{E}[\text{mh } \{t_M\} \{t_H\} (\text{cons } e_H e'_H)]_M \rightarrow \mathcal{E}[\text{cons } (\text{mh } t_M t_H e_H) (\text{mh } \{t_M\} \{t_H\} e'_H)] \\
& \mathcal{E}[\text{mh } (t_M \rightarrow t'_M) (t_H \rightarrow t'_H) (\lambda x_H : t''_H . e_H)]_M \rightarrow \\
& \quad \mathcal{E}[\lambda x_M : t_M . \text{mh } t'_M t'_H ((\lambda x_H : t''_H . e_H) (\text{hm } t_H t_M x_M))] \\
& \mathcal{E}[\text{mh } (\forall u_M . t_M) (\forall u_H . t_H) (\Lambda u'_H . e_H)]_M \rightarrow \mathcal{E}[\Lambda u_M . \text{mh } t_M t_H [\text{L}/u_H] e_H [\text{L}/u'_H]]
\end{aligned}$$

**Figure 2.14:** ML-Haskell operational semantics

$$\begin{aligned}
& \mathcal{E}[\mathbf{ms} \ N \ \bar{n}]_M \rightarrow \mathcal{E}[\bar{n}] \\
& \mathcal{E}[\mathbf{ms} \ N \ w_S]_M \rightarrow \mathcal{E}[\mathbf{wrong} \ N \ \text{“Not a number”}] \ (w_S \neq \bar{n}) \\
& \mathcal{E}[\mathbf{ms} \ \{k_M\} \ \mathbf{nil}]_M \rightarrow \mathcal{E}[\mathbf{nil} \ [k_M]] \\
& \mathcal{E}[\mathbf{ms} \ \{k_M\} \ (\mathbf{cons} \ v_S \ v'_S)]_M \rightarrow \mathcal{E}[\mathbf{cons} \ (\mathbf{ms} \ k_M \ v_S) \ (\mathbf{ms} \ \{k_M\} \ v'_S)] \\
& \mathcal{E}[\mathbf{ms} \ \{k_M\} \ w_S]_M \rightarrow \mathcal{E}[\mathbf{wrong} \ [\{k_M\}] \ \text{“Not a list”}] \\
& \quad (w_S \neq \mathbf{nil} \text{ and } w_S \neq \mathbf{cons} \ v_S \ v'_S) \\
& \mathcal{E}[\mathbf{ms} \ (b \diamond t_M) \ (\mathbf{sm} \ (b \diamond t_M) \ v_M)]_M \rightarrow \mathcal{E}[v_M] \\
& \mathcal{E}[\mathbf{ms} \ (b \diamond t_M) \ w_S]_M \rightarrow \mathcal{E}[\mathbf{wrong} \ [b \diamond t_M] \ \text{“Brand mismatch”}] \\
& \quad (w_S \neq \mathbf{sm} \ (b \diamond t_M) \ e_M) \\
& \mathcal{E}[\mathbf{ms} \ (k_M \rightarrow k'_M) \ (\lambda x_S. e_S)]_M \rightarrow \\
& \quad \mathcal{E}[\lambda x_M : [k_M]. \mathbf{ms} \ k'_M \ ((\lambda x_S. e_S) \ (\mathbf{sm} \ k_M \ x_M))] \\
& \mathcal{E}[\mathbf{ms} \ (k_M \rightarrow k'_M) \ w_S]_M \rightarrow \mathcal{E}[\mathbf{wrong} \ [k_M \rightarrow k'_M] \ \text{“Not a function”}] \\
& \quad (w_S \neq \lambda x_S. e_S) \\
& \mathcal{E}[\mathbf{ms} \ (\forall u_M. k_M) \ w_S]_M \rightarrow \mathcal{E}[\Lambda u_M. \mathbf{ms} \ k_M \ w_S]
\end{aligned}$$

**Figure 2.15: ML-Scheme operational semantics**

$$\begin{aligned}
e_S &= x_S \mid v_S \mid e_S e_S \mid o e_S e_S \mid p e_S \mid \mathbf{if0} e_S e_S e_S \mid \mathbf{cons} e_S e_S \mid f e_S \\
&\quad \mathbf{wrong} \textit{string} \mid \mathbf{sm} k_M e_M \\
v_S &= w_S \mid \mathbf{sh} k_H e_H \\
w_S &= \lambda x_S. e_S \mid \bar{n} \mid \mathbf{nil} \mid \mathbf{cons} v_S v_S \mid \mathbf{sh} (b \diamond t_H) e_H \mid \mathbf{sm} (b \diamond t_M) w_M \\
o &= + \mid - \\
f &= \mathbf{hd} \mid \mathbf{tl} \\
p &= \mathbf{fun?} \mid \mathbf{list?} \mid \mathbf{null?} \mid \mathbf{num?} \\
E_S &= U_S \mid \mathbf{sh} k_H E_H \\
U_S &= []_S \mid E_S e_S \mid w_S U_S \mid o E_S e_S \mid o w_S E_S \mid p E_S \mid \mathbf{if0} E_S e_S e_S \\
&\quad \mathbf{cons} U_S e_S \mid \mathbf{cons} v_S U_S \mid f E_S \mid \mathbf{sm} k_M E_M
\end{aligned}$$

**Figure 2.16:** Scheme grammar and evaluation contexts

$$\begin{array}{c}
\overline{\vdash_S \text{TST}} \\
\\
\frac{\Gamma, x_S : \text{TST} \vdash_S e_S : \text{TST}}{\Gamma \vdash_S \lambda x_S. e_S : \text{TST}} \quad \overline{\vdash_S \bar{n} : \text{TST}} \quad \overline{\vdash_S \text{nil} : \text{TST}} \\
\frac{\Gamma \vdash_S e_S : \text{TST} \quad \Gamma \vdash_S e'_S : \text{TST}}{\Gamma \vdash_S \text{cons } e_S e'_S : \text{TST}} \quad \overline{\Gamma, x_S : \text{TST} \vdash_S x_S : \text{TST}} \\
\frac{\Gamma \vdash_S e_S : \text{TST} \quad \Gamma \vdash_S e'_S : \text{TST}}{\Gamma \vdash_H e_S e'_S : \text{TST}} \quad \frac{\Gamma \vdash_S e_S : \text{TST}}{\Gamma \vdash_S f e_S : \text{TST}} \\
\frac{\Gamma \vdash_S e_S : \text{TST} \quad \Gamma \vdash_S e'_S : \text{TST}}{\Gamma \vdash_S o e_S e'_S : \text{TST}} \quad \frac{\Gamma \vdash_S e_S : \text{TST}}{\Gamma \vdash_S p e_S : \text{TST}} \\
\frac{\Gamma \vdash_S e_S : \text{TST} \quad \Gamma \vdash_S e'_S : \text{TST} \quad \Gamma \vdash_S e''_S : \text{TST}}{\Gamma \vdash_S \text{if0 } e_S e'_S e''_S : \text{TST}} \quad \overline{\vdash_S \text{wrong string} : \text{TST}} \\
\frac{\Gamma \vdash_H [k_H] \quad \Gamma \vdash_H e_H : t_H \quad [k_H] = t_H}{\Gamma \vdash_S \text{sh } k_H e_H : \text{TST}} \quad \frac{\Gamma \vdash_M [k_M] \quad \Gamma \vdash_M e_M : t_M \quad [k_M] = t_M}{\Gamma \vdash_S \text{sm } k_M e_M : \text{TST}}
\end{array}$$

**Figure 2.17: Scheme typing rules**

$$\begin{aligned}
& \mathcal{E}[(\lambda x_S. e_S) v_S]_S \rightarrow \mathcal{E}[e_S[v_S/x_S]] \\
& \mathcal{E}[w_S v_S]_S \rightarrow \mathcal{E}[\mathbf{wrong} \text{ "Not a function"}] \ (w_S \neq \lambda x_S. e_S) \\
& \mathcal{E}[+ \bar{n} \bar{n}']_S \rightarrow \mathcal{E}[\overline{n + n'}] \\
& \mathcal{E}[- \bar{n} \bar{n}']_S \rightarrow \mathcal{E}[\overline{\max(n - n', 0)}] \\
& \mathcal{E}[o w_S w'_S]_S \rightarrow \mathcal{E}[\mathbf{wrong} \text{ "Not a number"}] \ (w_S \neq \bar{n} \text{ or } w'_S \neq \bar{n}) \\
& \mathcal{E}[\text{if0 } \bar{0} e_S e'_S]_S \rightarrow \mathcal{E}[e_S] \\
& \mathcal{E}[\text{if0 } \bar{n} e_S e'_S]_S \rightarrow \mathcal{E}[e'_S] \ (n \neq 0) \\
& \mathcal{E}[\text{if0 } w_S e_S e'_S]_S \rightarrow \mathcal{E}[\mathbf{wrong} \text{ "Not a number"}] \ (w_S \neq \bar{n}) \\
& \mathcal{E}[f \text{ nil}]_S \rightarrow \mathcal{E}[\mathbf{wrong} \text{ "Empty list"}] \\
& \mathcal{E}[\text{hd} (\text{cons } v_S v'_S)]_S \rightarrow \mathcal{E}[v_S] \\
& \mathcal{E}[\text{tl} (\text{cons } v_S v'_S)]_S \rightarrow \mathcal{E}[v'_S] \\
& \mathcal{E}[f w_S]_S \rightarrow \mathcal{E}[\mathbf{wrong} \text{ "Not a list"}] \ (w_S \neq \text{nil} \text{ and } w_S \neq \text{cons } v_S v'_S) \\
& \mathcal{E}[\text{fun? } (\lambda x_S. e_S)]_S \rightarrow \mathcal{E}[\bar{0}] \\
& \mathcal{E}[\text{fun? } w_S]_S \rightarrow \mathcal{E}[\bar{1}] \ (w_S \neq \lambda x_S. e_S) \\
& \mathcal{E}[\text{list? nil}]_S \rightarrow \mathcal{E}[\bar{0}] \\
& \mathcal{E}[\text{list? } (\text{cons } v_S v'_S)]_S \rightarrow \mathcal{E}[\bar{0}] \\
& \mathcal{E}[\text{list? } w_S]_S \rightarrow \mathcal{E}[\bar{1}] \ (w_S \neq \text{nil} \text{ and } w_S \neq \text{cons } v_S v'_S) \\
& \mathcal{E}[\text{null? nil}]_S \rightarrow \mathcal{E}[\bar{0}] \\
& \mathcal{E}[\text{null? } w_S]_S \rightarrow \mathcal{E}[\bar{1}] \ (w_S \neq \text{nil}) \\
& \mathcal{E}[\text{num? } \bar{n}]_S \rightarrow \mathcal{E}[\bar{0}] \\
& \mathcal{E}[\text{num? } w_S]_S \rightarrow \mathcal{E}[\bar{1}] \ (w_S \neq \bar{n}) \\
& \mathcal{E}[\mathbf{wrong } string]_S \rightarrow \mathbf{Error: } string
\end{aligned}$$

**Figure 2.18: Scheme operational semantics**

$$\begin{aligned}
\mathcal{E}[\text{sh } L \text{ (hm } L \ k_M \ w_M)]_S &\rightarrow \mathcal{E}[\text{wrong "Bad value"}] \\
\mathcal{E}[\text{sh } L \text{ (hs } L \ w_S)]_S &\rightarrow \mathcal{E}[w_S] \\
\mathcal{E}[\text{sh } N \ \bar{n}]_S &\rightarrow \mathcal{E}[\bar{n}] \\
\mathcal{E}[\text{sh } \{k_H\} \text{ (nil } t_H)]_S &\rightarrow \mathcal{E}[\text{nil}] \\
\mathcal{E}[\text{sh } \{k_H\} \text{ (cons } e_H \ e'_H)]_S &\rightarrow \mathcal{E}[\text{cons } (\text{sh } k_H \ e_H) \ (\text{sh } \{k_H\} \ e'_H)] \\
\mathcal{E}[\text{sh } (k_H \rightarrow k'_H) \ (\lambda x_H : t_H.e_H)]_S &\rightarrow \\
&\quad \mathcal{E}[\lambda x_S.\text{sh } k'_H \ ((\lambda x_H : t_H.e_H) \ (\text{hs } k_H \ x_S))] \\
\mathcal{E}[\text{sh } (\forall u_H.k_H) \ (\Lambda u'_H.e_H)]_S &\rightarrow \mathcal{E}[\text{sh } k_H[L/u_H] \ e_H[L/u'_H]]
\end{aligned}$$

**Figure 2.19: Scheme-Haskell operational semantics**

$$\begin{aligned}
\mathcal{E}[\mathbf{sm} \ L \ (\mathbf{mh} \ L \ k_H \ e_H)]_S &\rightarrow \mathcal{E}[\mathbf{wrong} \ \text{“Bad value”}] \\
\mathcal{E}[\mathbf{sm} \ L \ (\mathbf{ms} \ L \ w_S)]_S &\rightarrow \mathcal{E}[w_S] \\
\mathcal{E}[\mathbf{sm} \ N \ \bar{n}]_S &\rightarrow \mathcal{E}[\bar{n}] \\
\mathcal{E}[\mathbf{sm} \ \{k_M\} \ (\mathbf{nil} \ t_M)]_S &\rightarrow \mathcal{E}[\mathbf{nil}] \\
\mathcal{E}[\mathbf{sm} \ \{k_M\} \ (\mathbf{cons} \ v_M \ v'_M)]_S &\rightarrow \mathcal{E}[\mathbf{cons} \ (\mathbf{sm} \ k_M \ v_M) \ (\mathbf{sm} \ \{k_M\} \ v'_M)] \\
\mathcal{E}[\mathbf{sm} \ (k_M \rightarrow k'_M) \ (\lambda x_M : t_M.e_M)]_S &\rightarrow \\
&\quad \mathcal{E}[\lambda x_S. \mathbf{sm} \ k'_M \ ((\lambda x_M : t_M.e_M) \ (\mathbf{ms} \ k_M \ x_S))] \\
\mathcal{E}[\mathbf{sm} \ (\forall u_M.k_M) \ (\Lambda u'_M.e_M)]_S &\rightarrow \mathcal{E}[\mathbf{sm} \ k_M[\mathbf{L}/u_M] \ e_M[\mathbf{L}/u'_M]]
\end{aligned}$$

**Figure 2.20: Scheme-ML operational semantics**

$$\begin{aligned}
\lfloor L \rfloor &= L \\
\lfloor N \rfloor &= N \\
\lfloor u_H \rfloor &= u_H \\
\lfloor u_M \rfloor &= u_M \\
\lfloor \{k_H\} \rfloor &= \{\lfloor k_H \rfloor\} \\
\lfloor \{k_M\} \rfloor &= \{\lfloor k_M \rfloor\} \\
\lfloor k_H \rightarrow k_H \rfloor &= \lfloor k_H \rfloor \rightarrow \lfloor k_H \rfloor \\
\lfloor k_M \rightarrow k_M \rfloor &= \lfloor k_M \rfloor \rightarrow \lfloor k_M \rfloor \\
\lfloor \forall u_H. k_H \rfloor &= \forall u_H. \lfloor k_H \rfloor \\
\lfloor \forall u_M. k_M \rfloor &= \forall u_M. \lfloor k_M \rfloor \\
\lfloor b \diamond t_H \rfloor &= t_H \\
\lfloor b \diamond t_M \rfloor &= t_M
\end{aligned}$$

**Figure 2.21: Unbrand function**

$$\begin{aligned}
x &\dot{=} x \\
x &\dot{=} y \Rightarrow y \dot{=} x \\
x &\dot{=} y \text{ and } y \dot{=} z \Rightarrow x \dot{=} z \\
t_H &\dot{=} L \\
t_M &\dot{=} L \\
t_H = t_M &\Rightarrow t_H \dot{=} t_M
\end{aligned}$$

**Figure 2.22: Lump equality relation**



# Chapter 3

## Proof of Type Soundness

Proving the progress of expressions and the preservation of types proves the type soundness of the model of computation. Progress ensures that a well-typed, closed expression is either an unforced value, reducible to another expression, or reducible to an error. Preservation ensures that if a well-typed expression reduces to another expression, the other expression is well-typed and has the same type. The proof extends the proof by Kinghorn [?], which was based on proofs by Pierce [?] and Matthews and Findler [?]. Cases common to two or more languages are elided for brevity.

### 3.1 Proof of Expression Progress

Progress will be proven by structural induction on a well-typed, closed expression of each syntactic form. In each case, the expression will be proven to be either an unforced value, reducible to another expression, or reducible to an error. The reduction of a subexpression is the reduction of its parent expression. If a subexpression reduces to an error, its parent expression reduces to the er-

ror. In some cases, the syntactic form of a subexpression must be determined to reduce its parent expression. Determining the unique type of a subexpression determines its syntactic form.

Inverting the typing rules enables the syntactic forms of well-typed expressions to determine the types of their subexpressions.

**Lemma 1.** *Inversion of the Typing Relation*

*The syntactic forms of well-typed expressions determine the types of their subexpressions. ML cases are omitted because they mirror those of Haskell. Straight-forward Scheme cases are omitted because in those cases well-typed Scheme expressions and subexpressions have the type TST.*

1. If  $\Gamma \vdash_H \lambda x_H : t_H. e_H : t'_H$  then  $t'_H = t_H \rightarrow t''_H$ ,  $\Gamma \vdash_H t_H$ , and  $\Gamma, x_H : t_H \vdash_H e_H : t''_H$ .
2. If  $\Gamma \vdash_H \Lambda u_H. e_H : t_H$  then  $t_H = \forall u_H. t'_H$  and  $\Gamma, u_H \vdash_H e_H : t'_H$ .
3. If  $\vdash_H \bar{n} : t_H$  then  $t_H = \mathbf{N}$ .
4. If  $\Gamma \vdash_H \mathbf{nil} : t_H$  then  $t_H = \{\mathbf{nil}\}$  and  $\Gamma \vdash_H \mathbf{nil}$ .
5. If  $\Gamma \vdash_H \mathbf{cons} e_H e'_H : t_H$  then  $t_H = \{t'_H\}$ ,  $\Gamma \vdash_H e_H : t'_H$ , and  $\Gamma \vdash_H e'_H : \{t'_H\}$ .
6.  $\Gamma, x_H : t_H \vdash_H x_H : t_H$ .
7. If  $\Gamma \vdash_H e_H e'_H : t_H$  then  $\Gamma \vdash_H e_H : t'_H \rightarrow t_H$  and  $\Gamma \vdash_H e'_H : t'_H$ .
8. If  $\Gamma \vdash_H \mathbf{fix} e_H : t_H$  then  $\Gamma \vdash_H e_H : t_H \rightarrow t_H$ .
9. If  $\Gamma \vdash_H e_H \langle t_H \rangle : t'_H$  then  $t'_H = t''_H[t_H/u_H]$ ,  $\Gamma \vdash_H t_H$ , and  $\Gamma \vdash_H e_H : \forall u_H. t''_H$ .

10. If  $\Gamma \vdash_H \mathbf{hd} \ e_H : t_H$  then  $\Gamma \vdash_H e_H : \{t_H\}$ .
11. If  $\Gamma \vdash_H \mathbf{tl} \ e_H : t_H$  then  $t_H = \{t'_H\}$  and  $\Gamma \vdash_H e_H : \{t'_H\}$ .
12. If  $\Gamma \vdash_H o \ e_H \ e'_H : t_H$  then  $t_H = \mathbb{N}$ ,  $\Gamma \vdash_H e_H : \mathbb{N}$ , and  $\Gamma \vdash_H e'_H : \mathbb{N}$ .
13. If  $\Gamma \vdash_H \mathbf{null?} \ e_H : t_H$  then  $t_H = \mathbb{N}$  and  $\Gamma \vdash_H e_H : \{t'_H\}$ .
14. If  $\Gamma \vdash_H \mathbf{if0} \ e_H \ e'_H \ e''_H : t_H$  then  $\Gamma \vdash_H e_H : \mathbb{N}$ ,  $\Gamma \vdash_H e'_H : t_H$ ,  $\Gamma \vdash_H e''_H : t_H$ .
15. If  $\Gamma \vdash_H \mathbf{wrong} \ t_H \ \mathit{string} : t'_H$  then  $t'_H = t_H$ .
16. If  $\Gamma \vdash_H \mathbf{hm} \ t_H \ t_M \ e_M : t'_H$  and  $t_H \doteq t_M$  then  $t'_H = t_H$ ,  $\Gamma \vdash_H t_H$ ,  $\Gamma \vdash_M t_M$ ,  $\Gamma \vdash_M e_M : t'_M$ , and  $t_M = t'_M$ .
17. If  $\Gamma \vdash_H \mathbf{hs} \ k_H \ e_S : t_H$  then  $t_H = \lfloor k_H \rfloor$ ,  $\Gamma \vdash_H \lfloor k_H \rfloor$ , and  $\Gamma \vdash_S e_S : \mathbf{TST}$ .
18. If  $\Gamma \vdash_S \mathbf{sh} \ k_H \ e_H : \mathbf{TST}$  then  $\Gamma \vdash_H \lfloor k_H \rfloor$ ,  $\Gamma \vdash_H e_H : t_H$ , and  $\lfloor k_H \rfloor = t_H$ .
19. If  $\Gamma \vdash_S \mathbf{sm} \ k_M \ e_M : \mathbf{TST}$  then  $\Gamma \vdash_M \lfloor k_M \rfloor$ ,  $\Gamma \vdash_M e_M : t_M$ , and  $\lfloor k_M \rfloor = t_M$ .

*Proof.* Immediate from the typing rules.

□

Well-typed Haskell and ML expressions have unique types.

**Lemma 2.** *Uniqueness of Types*

*If  $e_H$ ,  $e_M$ , and  $e_S$  are well-typed then they have only one type.*

*Proof.* Straightforward structural induction on  $e_H$ ,  $e_M$ , and  $e_S$  using the induction hypothesis and the inversion lemma.

□

The types of Haskell and ML values determine their syntactic forms.

**Lemma 3.** *Canonical Forms*

*The syntactic forms of unforced values for each type.*

1. If  $\Gamma \vdash_H v_H : \mathbf{L}$  then  $v_H \in \{\mathbf{hm} \ \mathbf{L} \ t_M \ w_M, \mathbf{hs} \ \mathbf{L} \ w_S\}$ .
2. If  $\Gamma \vdash_H v_H : \mathbf{N}$  then  $v_H = \bar{n}$ .
3. If  $\Gamma \vdash_H v_H : \{t_H\}$  then  $v_H \in \{\mathbf{nil} \ t_H, \mathbf{cons} \ e_H \ e'_H\}$ .
4. If  $\Gamma \vdash_H v_H : t_H \rightarrow t'_H$  then  $v_H = \lambda x_H : t_H. e_H$ .
5. If  $\Gamma \vdash_H v_H : \forall u_H. t_H$  then  $v_H = \Lambda u_H. e_H$ .
6. If  $\Gamma \vdash_M w_M : \mathbf{L}$  then  $w_M \in \{\mathbf{mh} \ \mathbf{L} \ t_M \ w_M, \mathbf{ms} \ \mathbf{L} \ w_S\}$ .
7. If  $\Gamma \vdash_M w_M : \mathbf{N}$  then  $w_M = \bar{n}$ .
8. If  $\Gamma \vdash_M w_M : \{t_M\}$  then  $w_M \in \{\mathbf{nil} \ t_M, \mathbf{cons} \ v_M \ v'_M\}$ .
9. If  $\Gamma \vdash_M w_M : t_M \rightarrow t'_M$  then  $w_M = \lambda x_H : t_M. e_H$ .
10. If  $\Gamma \vdash_M w_M : \forall u_M. t_M$  then  $w_M = \Lambda u_M. e_H$ .

*Proof.* Immediate from the definitions of unforced values and the typing relations.

□

**Theorem 1.** *Haskell Progress*

*If  $\vdash_H e_H : t_H$  then  $e_H$  is an unforced value or  $e_H \rightarrow e'_H$  or  $e_H \rightarrow \mathbf{Error}$ : string.*

*Proof.* By structural induction on  $e_H$  and theorems 2 and 3.

**Case 1.1.**  $x_H$

*Cannot occur because  $e_H$  is closed.*

**Case 1.2.**  $\lambda x_H : t_H.e_H$

*$\lambda x_H : t_H.e_H$  is an unforced value.*

**Case 1.3.**  $\Lambda u_H.e_H$

*$\Lambda u_H.e_H$  is an unforced value.*

**Case 1.4.**  $\bar{n}$

*$\bar{n}$  is an unforced value.*

**Case 1.5.**  $\text{nil } t_H$

*$\text{nil } t_H$  is an unforced value.*

**Case 1.6.**  $\text{cons } e_H e'_H$

*$\text{cons } e_H e'_H$  is an unforced value.*

**Case 1.7.**  $e_H e'_H$

*$e_H$  is an unforced value or  $e_H \rightarrow e''_H$  or  $e_H \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_H$  is an unforced value then  $\vdash_H e_H : t_H \rightarrow t'_H$  by lemmas 1 and 2 and  $e_H = \lambda x_H : t_H.e''_H$  by lemma 3.  $(\lambda x_H : t_H.e''_H) e'_H \rightarrow e''_H[e'_H/x_H]$ . If  $e_H \rightarrow e''_H$  then  $e_H e'_H \rightarrow e''_H e'_H$ . If  $e_H \rightarrow \mathbf{Error}$ : string then  $e_H e'_H \rightarrow \mathbf{Error}$ : string.*

**Case 1.8.**  $e_H \langle t_H \rangle$

*$e_H$  is an unforced value or  $e_H \rightarrow e'_H$  or  $e_H \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_H$  is an unforced value then  $\vdash_H e_H : \forall u_H.t'_H$  by lemmas 1 and 2 and  $e_H = \Lambda u_H.e'_H$  by lemma 3.  $(\Lambda u_H.e'_H) \langle t_H \rangle \rightarrow e'_H[t_H/u_H]$ . If  $e_H \rightarrow e'_H$  then  $e_H \langle t_H \rangle \rightarrow e'_H \langle t_H \rangle$ . If  $e_H \rightarrow \mathbf{Error}$ : string then  $e_H \langle t_H \rangle \rightarrow \mathbf{Error}$ : string.*

**Case 1.9.  $\text{fix } e_H$**

$e_H$  is an unforced value or  $e_H \rightarrow e'_H$  or  $e_H \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_H$  is an unforced value then  $\vdash_H e_H : t_H \rightarrow t_H$  by lemmas 1 and 2 and  $e_H = \lambda x_H : t_H. e'_H$  by lemma 3.  $\text{fix } (\lambda x_H : t_H. e'_H) \rightarrow e'_H[\text{fix } (\lambda x_H : t_H. e'_H)/x_H]$ . If  $e_H \rightarrow e'_H$  then  $\text{fix } e_H \rightarrow \text{fix } e'_H$ . If  $e_H \rightarrow \mathbf{Error}$ : string then  $\text{fix } e_H \rightarrow \mathbf{Error}$ : string.

**Case 1.10.  $o \ e_H \ e'_H$**

$e_H$  is an unforced value or  $e_H \rightarrow e''_H$  or  $e_H \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_H$  is an unforced value then  $\vdash_H e_H : \mathbb{N}$  by lemmas 1 and 2 and  $e_H = \bar{n}$  by lemma 3. If  $e_H \rightarrow e''_H$  then  $o \ e_H \ e'_H \rightarrow o \ e''_H \ e'_H$ . If  $e_H \rightarrow \mathbf{Error}$ : string then  $o \ e_H \ e'_H \rightarrow \mathbf{Error}$ : string.  $e'_H$  is an unforced value or  $e'_H \rightarrow e''_H$  or  $e'_H \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e'_H$  is an unforced value then  $\vdash_H e'_H : \mathbb{N}$  by lemmas 1 and 2 and  $e'_H = \bar{n}'$  by lemma 3. If  $e'_H \rightarrow e''_H$  and  $e_H$  is an unforced value then  $o \ e_H \ e'_H \rightarrow o \ e_H \ e''_H$ . If  $e'_H \rightarrow \mathbf{Error}$ : string and  $e_H$  is an unforced value then  $o \ e_H \ e'_H \rightarrow \mathbf{Error}$ : string.  $+ \ \bar{n} \ \bar{n}' \rightarrow \overline{n + n'}$ .  $- \ \bar{n} \ \bar{n}' \rightarrow \overline{\max(n - n', 0)}$ .

**Case 1.11.  $\text{if0 } e_H \ e'_H \ e''_H$**

$e_H$  is an unforced value or  $e_H \rightarrow e'''_H$  or  $e_H \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_H$  is an unforced value then  $\vdash_H e_H : \mathbb{N}$  by lemmas 1 and 2 and  $e_H = \bar{n}$  by lemma 3.  $\text{if0 } \bar{0} \ e'_H \ e''_H \rightarrow e'_H$ .  $\text{if0 } \bar{n} \ e'_H \ e''_H \rightarrow e''_H$  ( $n \neq 0$ ). If  $e_H \rightarrow e'''_H$  then  $\text{if0 } e_H \ e'_H \ e''_H \rightarrow \text{if0 } e'''_H \ e'_H \ e''_H$ . If  $e_H \rightarrow \mathbf{Error}$ : string then  $\text{if0 } e_H \ e'_H \ e''_H \rightarrow \mathbf{Error}$ : string.

**Case 1.12.  $f \ e_H$**

$e_H$  is an unforced value or  $e_H \rightarrow e'_H$  or  $e_H \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_H$  is an unforced value then  $\vdash_H e_H : \{t_H\}$  by lemmas 1 and 2 and

$e_H \in \{\text{nil } t_H, \text{cons } e'_H e''_H\}$  by lemma 3.  $\text{hd } (\text{nil } t_H) \rightarrow \text{wrong } t_H$  “Empty list”.  
 $\text{tl } (\text{nil } t_H) \rightarrow \text{wrong } \{t_H\}$  “Empty list”.  $\text{hd } (\text{cons } e'_H e''_H) \rightarrow e'_H$ .  $\text{tl } (\text{cons } e'_H e''_H) \rightarrow e''_H$ .  
If  $e_H \rightarrow e'_H$  then  $f e_H \rightarrow f e'_H$ . If  $e_H \rightarrow \mathbf{Error: string}$  then  $f e_H \rightarrow \mathbf{Error: string}$ .

**Case 1.13.**  $\text{null? } e_H$

$e_H$  is an unforced value or  $e_H \rightarrow e'_H$  or  $e_H \rightarrow \mathbf{Error: string}$  by the induction hypothesis. If  $e_H$  is an unforced value then  $\vdash_H e_H : \{t_H\}$  by lemmas 1 and 2 and  $e_H \in \{\text{nil } t_H, \text{cons } e'_H e''_H\}$  by lemma 3.  $\text{null? } (\text{nil } t_H) \rightarrow \bar{0}$ .  
 $\text{null? } (\text{cons } e'_H e''_H) \rightarrow \bar{1}$ . If  $e_H \rightarrow e'_H$  then  $\text{null? } e_H \rightarrow \text{null? } e'_H$ . If  $e_H \rightarrow \mathbf{Error: string}$  then  $\text{null? } e_H \rightarrow \mathbf{Error: string}$ .

**Case 1.14.**  $\text{wrong } t_H$  string

$\text{wrong } t_H$  string  $\rightarrow \mathbf{Error: string}$ .

**Case 1.15.**  $\text{hm } t_H t_M e_M$

$e_M$  is an unforced value or  $e_M \rightarrow e'_M$  or  $e_M \rightarrow \mathbf{Error: string}$  by the induction hypothesis. If  $e_M \rightarrow e'_M$  then  $\text{hm } t_H t_M e_M \rightarrow \text{hm } t_H t_M e'_M$ . If  $e_M \rightarrow \mathbf{Error: string}$  then  $\text{hm } t_H t_M e_M \rightarrow \mathbf{Error: string}$ . If  $e_M$  is an unforced value then  $t_H$  and  $t_M$  determine the reduction of  $\text{hm } t_H t_M e_M$ .

**Case 1.15.1.**  $t_H = L$

$\text{hm } L t_M e_M$  is an unforced value.

**Case 1.15.2.**  $t_H \neq L$  and  $t_M = L$

$\vdash_H e_M : L$  by lemmas 1 and 2 and  $e_M \in \{\text{mh } L t'_H e_H, \text{ms } L w_S\}$  by lemma 3.  
 $\text{hm } t_H L (\text{mh } L t'_H e_H) \rightarrow e_H (t_H = t'_H)$ .  $\text{hm } t_H L (\text{mh } L t'_H e_H) \rightarrow e_H (t_H \neq t'_H)$ .  
 $\text{hm } t_H L (\text{ms } L w_S) \rightarrow \text{wrong } t_H$  “Bad value”.

**Case 1.15.3.**  $t_H = N$  and  $t_M = N$

$\vdash_M e_M : \mathbb{N}$  by lemmas 1 and 2 and  $e_M = \bar{n}$  by lemma 3.  $\text{hm } \mathbb{N} \ \bar{n} \rightarrow \bar{n}$ .

**Case 1.15.4.**  $t_H = \{t'_H\}$  and  $t_M = \{t'_M\}$

$\vdash_M e_M : \{t''_M\}$  by lemmas 1 and 2 and  $e_M \in \{\text{nil } t''_M, \text{cons } v_M \ v'_M\}$  by lemma 3.  $\text{hm } \{t'_H\} \ \{t'_M\} \ (\text{nil } t''_M) \rightarrow \text{nil } t'_H$ .  $\text{hm } \{t'_H\} \ \{t'_M\} \ (\text{cons } v_M \ v'_M) \rightarrow \text{cons } (\text{hm } t'_H \ t'_M \ v_M) \ (\text{hm } \{t'_H\} \ \{t'_M\} \ v'_M)$ .

**Case 1.15.5.**  $t_H = t'_H \rightarrow t''_H$  and  $t_M = t'_M \rightarrow t''_M$

$\vdash_M e_M : t'''_M \rightarrow t''''_M$  by lemmas 1 and 2 and  $e_M = \lambda x_M : t'''_M.e'_M$  by lemma 3.  $\text{hm } (t'_H \rightarrow t''_H) \ (t'_M \rightarrow t''_M) \ (\lambda x_M : t'''_M.e'_M) \rightarrow \lambda x_H : t'_H.\text{hm } t''_H \ t'_M \ ((\lambda x_M : t'''_M.e'_M) \ (\text{mh } t'_M \ t'_H \ x_H))$ .

**Case 1.15.6.**  $t_H = \forall u_H.t'_H$  and  $t_M = \forall u_M.t'_M$

$\vdash_M e_M : \forall u'_M.t''_M$  by lemmas 1 and 2 and  $e_M = \Lambda u'_M.e'_M$  by lemma 3.  $\text{hm } (\forall u_H.t'_H) \ (\forall u_M.t'_M) \ (\Lambda u'_M.e'_M) \rightarrow \Lambda u_H.\text{hm } t'_H \ t'_M[\text{L}/u_M] \ e'_M[\text{L}/u'_M]$ .

**Case 1.16.**  $\text{hs } k_H \ e_S$

$e_S$  is an unforced value or  $e_S \rightarrow e'_S$  or  $e_S \rightarrow \text{Error: string}$  by the induction hypothesis. If  $e_S \rightarrow e'_S$  then  $\text{hs } k_H \ e_S \rightarrow \text{hs } k_H \ e'_S$ . If  $e_S \rightarrow \text{Error: string}$  then  $\text{hs } k_H \ e_S \rightarrow \text{Error: string}$ . If  $e_S$  is an unforced value then  $k_H$  determines the reduction of  $\text{hs } k_H \ e_S$ .

**Case 1.16.1.** L

$\text{hs } \text{L } e_S$  is an unforced value.

**Case 1.16.2.** N

$\text{hs } \mathbb{N} \ \bar{n} \rightarrow \bar{n}$ .  $\text{hs } \mathbb{N} \ e_S \rightarrow \text{wrong } \mathbb{N} \ \text{“Not a number”} \ (e_S \neq \bar{n})$ .

**Case 1.16.3.**  $\{k'_H\}$



$\text{hs } \{k'_H\} \text{ nil} \rightarrow \text{nil } [k'_H]. \text{hs } \{k'_H\} (\text{cons } v_S v'_S) \rightarrow \text{cons } (\text{hs } k_H v_S) (\text{hs } \{k'_H\} v'_S).$   
 $\text{hs } \{k'_H\} e_S \rightarrow \text{wrong } [k'_H]$  “Not a list” ( $e_S \neq \text{nil}$  and  $e_S \neq \text{cons } v_S v'_S$ ).

**Case 1.16.4.**  $b \diamond t_H$

$\text{hs } (b \diamond t_H) (\text{sh } (b \diamond t_H) e_H) \rightarrow e_H. \text{hs } (b \diamond t_H) e_S \rightarrow \text{wrong } t_H$  “Brand mismatch”  
 $(e_S \neq \text{sh } (b \diamond t_H) e_H).$

**Case 1.16.5.**  $k'_H \rightarrow k''_H$

$\text{hs } (k'_H \rightarrow k''_H) (\lambda x_S. e_S) \rightarrow \lambda x_H : [k'_H]. \text{hs } k''_H ((\lambda x_S. e_S) (\text{sh } k'_H x_H)).$   
 $\text{hs } (k'_H \rightarrow k''_H) e_S \rightarrow \text{wrong } [k'_H \rightarrow k''_H]$  “Not a function” ( $e_S \neq \lambda x_S. e_S$ ).

**Case 1.16.6.**  $\forall u_H. k'_H$

$\text{hs } (\forall u_H. k'_H) e_S \rightarrow \Lambda u_H. \text{hs } k'_H e_S.$

□

**Theorem 2.** *ML Progress*

*If  $\vdash_M e_M : t_M$  then  $e_M$  is an unforced value or  $e_M \rightarrow e'_M$  or  $e_M \rightarrow \mathbf{Error}$ : string.*

*Proof.* By structural induction on  $e_M$  and theorems 1 and 3. Cases similar to Haskell cases are omitted.

**Case 2.1.**  $\text{mh } t_M t_H e_H$

$e_H$  is an unforced value or  $e_H \rightarrow e'_H$  or  $e_H \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_H$  is an unforced value then  $t_M$  and  $t_H$  determine the reduction of  $\text{mh } t_M t_H e_H$ .

TODO

**Case 2.1.1.**

If  $e_H \rightarrow e'_H$  then  $\text{mh } t_M t_H e_H \rightarrow \text{mh } t_M t_H e'_H$ . If  $e_H \rightarrow \mathbf{Error}$ : string then  $\text{mh } t_M t_H e_H \rightarrow \mathbf{Error}$ : string.

**Case 2.2.**  $e_M e'_M$

$e_M$  is an unforced value or  $e_M \rightarrow e''_M$  or  $e_M \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_M$  is an unforced value then  $\vdash_M e_M : t_M \rightarrow t'_M$  by lemmas 1 and 2 and  $e_M = \lambda x_M : t_M. e''_M$  by lemma 3. If  $e_M \rightarrow e''_M$  then  $e_M e'_M \rightarrow e''_M e'_M$ . If  $e_M \rightarrow \mathbf{Error}$ : string then  $e_M e'_M \rightarrow \mathbf{Error}$ : string.  $e'_M$  is an unforced value or  $e'_M \rightarrow e''_M$  or  $e'_M \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e'_M \rightarrow e''_M$  and  $e_M$  is an unforced value then  $e_M e'_M \rightarrow e_M e''_M$ . If  $e'_M \rightarrow \mathbf{Error}$ : string and  $e_M$  is an unforced value then  $e_M e'_M \rightarrow \mathbf{Error}$ : string. If  $e_M$  and  $e'_M$  are unforced values then  $e_M \rightarrow e'_M$ .

**Case 2.3.**  $\text{cons } e_M e'_M$

$e_M$  is an unforced value or  $e_M \rightarrow e''_M$  or  $e_M \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_M \rightarrow e''_M$  then  $\text{cons } e_M e'_M \rightarrow \text{cons } e''_M e'_M$ . If  $e_M \rightarrow \mathbf{Error}$ : string then  $\text{cons } e_M e'_M \rightarrow \mathbf{Error}$ : string. If  $e'_M \rightarrow e''_M$  and  $e_M$  is an unforced value then  $\text{cons } e_M e'_M \rightarrow \text{cons } e''_M e'_M$ . If  $e'_M \rightarrow \mathbf{Error}$ : string and  $e_M$  is an unforced value then  $\text{cons } e_M e'_M \rightarrow \mathbf{Error}$ : string. If  $e_M$  and  $e'_M$  are unforced values then  $\text{cons } e_M e'_M$  is an unforced value.

□

**Theorem 3.** *Scheme Progress*

If  $\vdash_S e_S : \text{TST}$  then  $e_S$  is an unforced value or  $e_S \rightarrow e'_S$  or  $e_S \rightarrow \mathbf{Error}$ : string.

*Proof.* By structural induction on  $e_S$  and theorems 1 and 2. Cases similar to Haskell and ML cases are omitted.

**Case 3.1.**  $e_S = SH^T e_H$

$SH^T e_H$  is an unforced value.

TODO

**Case 3.2.**  $\text{sm } k_M e_M$

$e_M$  is an unforced value or  $e_M \rightarrow e'_M$  or  $e_M \rightarrow \mathbf{Error}$ : string by the induction hypothesis. If  $e_M \rightarrow e'_M$  then  $\text{sm } k_M e_M \rightarrow \text{sm } k_M e'_M$ . If  $e_M \rightarrow \mathbf{Error}$ : string then  $\text{sm } k_M e_M \rightarrow \mathbf{Error}$ : string. If  $e_M$  is an unforced value then  $k_M$  determines the reduction of  $\text{sm } k_M e_M$ .

**Case 3.2.1.** L

$e_M \in \{\text{mh L } k_H e_H, \text{ms L } w_S\}$  by lemma 3.  $\text{sm L } (\text{mh L } k_H e_H) \rightarrow \mathbf{wrong}$  “Bad value”.  
 $\text{sm L } (\text{ms L } w_S) \rightarrow w_S$ .

**Case 3.2.2.** N

$e_M = \bar{n}$  by lemma 3.  $\text{sm N } \bar{n} \rightarrow \bar{n}$ .

**Case 3.2.3.**  $\{k'_M\}$

$e_M \in \{\mathbf{nil } t_M, \text{cons } v_M v'_M\}$  by lemma 3.  $\text{sm } \{k'_M\} (\mathbf{nil } t_M) \rightarrow \mathbf{nil}$ .  
 $\text{sm } \{k'_M\} (\text{cons } v_M v'_M) \rightarrow \text{cons } (\text{sm } k'_M v_M) (\text{sm } \{k'_M\} v'_M)$ .

**Case 3.2.4.**  $k'_M \rightarrow k''_M$

$e_M = \lambda x_M : t_M.e_M$  by lemma 3.  $\text{sm } (k'_M \rightarrow k''_M) (\lambda x_M : t_M.e_M) \rightarrow \lambda x_S. \text{sm } k''_M ((\lambda x_M : t_M.e_M) (\text{ms } k'_M x_S))$ .

**Case 3.2.5.**  $\forall u_M.k'_M$

$e_M = \Lambda u'_M.e'_M$  by lemma 3.  $\text{sm } (\forall u_M.k'_M) (\Lambda u'_M.e'_M) \rightarrow \text{sm } k'_M[\mathbf{L}/u_M] e'_M[\mathbf{L}/u'_M]$ .

**Case 3.2.6.**  $b \diamond t_M$

$\text{sm } (b \diamond t_M) e_M$  is an unforced value.

□

## 3.2 Proof of Type Preservation

Preservation will be proven by cases on the reduction rules. In each case, the right side will be proven to be well-typed and have the same type as the left side. Lemmas 1 and 2 determine the types of the left side and its subexpressions and the type of the right side. Some reduction rules use expression and type substitutions.

If  $e_H$  is substituted for free occurrences of  $x_H$  within  $e'_H$  and  $e_H$  and  $x_H$  have the same type then the result has the same type as  $e'_H$ .

**Lemma 4.** *Expression Substitution Preservation*

If  $\Gamma, x_H : t_H \vdash_H e_H : t'_H$  and  $\Gamma \vdash_H e'_H : t_H$  then  $\Gamma \vdash_H e_H[e'_H/x_H] : t'_H$ . If  $\Gamma, x_M : t_M \vdash_M e_M : t'_M$  and  $\Gamma \vdash_M e'_M : t_M$  then  $\Gamma \vdash_M e_M[e'_M/x_M] : t'_M$ . If  $\Gamma, x_S : \text{TST} \vdash_S e_S : \text{TST}$  and  $\Gamma \vdash_S e'_S : \text{TST}$  then  $\Gamma \vdash_S e_S[e'_S/x_S] : \text{TST}$ .

*Proof.* By structural induction.

□

If  $t_H$  is substituted for free occurrences of  $u_H$  within  $e_H$  of type  $t'_H$ , the type of the result is  $t_H$  substituted for free occurrences of  $u_H$  within  $t'_H$ .

**Lemma 5.** *Type Substitution Preservation*

If  $\Gamma, u_H \vdash_H e_H : t_H$  and  $\Gamma \vdash_H t'_H$  then  $\Gamma \vdash_H e_H[t'_H/u_H] : t_H[t'_H/u_H]$ . If  $\Gamma, u_M \vdash_M e_M : t_M$  and  $\Gamma \vdash_M t'_M$  then  $\Gamma \vdash_M e_M[t'_M/u_M] : t_M[t'_M/u_M]$ .

*Proof.* By structural induction. □

**Lemma 6.** *Evaluation Context Preservation*

If  $\vdash_H e_H : t_H$ ,  $\vdash_H e'_H : t_H$ , and  $\vdash_H \mathcal{E}[e_H]_H : t'_H$  then  $\vdash_H \mathcal{E}[e'_H]_H : t'_H$ . If  $\vdash_M e_M : t_M$ ,  $\vdash_M e'_M : t_M$ , and  $\vdash_M \mathcal{E}[e_M]_M : t'_M$  then  $\vdash_M \mathcal{E}[e'_M]_M : t'_M$ . If  $\vdash_S e_S : \text{TST}$ ,  $\vdash_S e'_S : \text{TST}$ , and  $\vdash_S \mathcal{E}[e_S]_S : \text{TST}$  then  $\vdash_S \mathcal{E}[e'_S]_S : \text{TST}$ .

*Proof.* By structural induction. □

**Theorem 4.** *Haskell Preservation*

If  $\Gamma \vdash_H e_H : t_H$  and  $\mathcal{E}[e_H]_H \rightarrow \mathcal{E}[e'_H]$  then  $\Gamma \vdash_H e'_H : t_H$ .

*Proof.* By cases on the reduction  $\mathcal{E}[e_H]_H \rightarrow \mathcal{E}[e'_H]$ , lemma 6, and theorems 5 and 6.

**Case 4.1.**  $\mathcal{E}[(\lambda x_H : t_H. e_H) e'_H]_H \rightarrow \mathcal{E}[e_H[e'_H/x_H]]$

$\Gamma \vdash_H (\lambda x_H : t_H. e_H) e'_H : t'_H$  by the premise and lemma 2.  $\Gamma \vdash_H \lambda x_H : t_H. e_H : t_H \rightarrow t'_H$ ;  $\Gamma, x_H : t_H \vdash_H e_H : t'_H$ ;  $\Gamma \vdash_H e'_H : t_H$ ; and  $\Gamma, x_H : t_H \vdash_H x_H : t_H$  by lemmas 1 and 2.  $\Gamma \vdash_H e_H[e'_H/x_H] : t'_H$  by lemma 4.

**Case 4.2.**  $\mathcal{E}[(\Lambda u_H. e_H)\langle t_H \rangle]_H \rightarrow \mathcal{E}[e_H[b \diamond t_H/u_H]]$

$\Gamma \vdash_H (\Lambda u_H. e_H)\langle t_H \rangle : t'_H$  by the premise and lemma 2.  $\Gamma \vdash_H \Lambda u_H. e_H : \forall u_H. t''_H$ ;  $\Gamma, u_H \vdash_H e_H : t''_H$ ; and  $t'_H = t''_H[t_H/u_H]$  by lemmas 1 and 2.  $\Gamma \vdash_H e_H[b \diamond t_H/u_H] : t''_H[t_H/u_H]$  by lemma 5.

**Case 4.3.**  $\mathcal{E}[\text{fix } (\lambda x_H : t_H. e_H)]_H \rightarrow \mathcal{E}[e_H[\text{fix } (\lambda x_H : t_H. e_H)/x_H]]$

$\Gamma \vdash_H \mathbf{fix} (\lambda x_H : t_H.e_H) : t'_H$  by the premise and lemma 2.  $\Gamma \vdash_H (\lambda x_H : t_H.e_H) : t_H \rightarrow t'_H$  and  $\Gamma, x_H : t_H \vdash_H e_H : t'_H$  by lemmas 1 and 2.  $\Gamma \vdash_H e_H[\mathbf{fix} (\lambda x_H : t_H.e_H)/x_H] : t'_H$  by lemma 4.

**Case 4.4.**  $\mathcal{E}[+ \bar{n} \bar{n}']_H \rightarrow \mathcal{E}[\overline{n + n'}]$

$\vdash_H + \bar{n} \bar{n}' : t_H$  by the premise and lemma 2.  $\vdash_H \bar{n} : \mathbb{N}, \vdash_H \bar{n}' : \mathbb{N}, t_H = \mathbb{N}$ , and  $\vdash_H \overline{n + n'} : \mathbb{N}$  by lemmas 1 and 2.

**Case 4.5.**  $\mathcal{E}[- \bar{n} \bar{n}']_H \rightarrow \mathcal{E}[\overline{\max(n - n', 0)}]$

$\vdash_H - \bar{n} \bar{n}' : t_H$  by the premise and lemma 2.  $\vdash_H \bar{n} : \mathbb{N}, \vdash_H \bar{n}' : \mathbb{N}, t_H = \mathbb{N}$ , and  $\vdash_H \overline{\max(n - n', 0)} : \mathbb{N}$  by lemmas 1 and 2.

**Case 4.6.**  $\mathcal{E}[\mathbf{if0} \bar{0} e_H e'_H]_H \rightarrow \mathcal{E}[e_H]$

$\Gamma \vdash_H \mathbf{if0} \bar{0} e_H e'_H : t_H$  by the premise and lemma 2.  $\Gamma \vdash_H e_H : t_H$  by lemmas 1 and 2.

**Case 4.7.**  $\mathcal{E}[\mathbf{if0} \bar{n} e_H e'_H]_H \rightarrow \mathcal{E}[e'_H]$  ( $n \neq 0$ )

$\Gamma \vdash_H \mathbf{if0} \bar{n} e_H e'_H : t_H$  by the premise and lemma 2.  $\Gamma \vdash_H e'_H : t_H$  by lemmas 1 and 2.

**Case 4.8.**  $\mathcal{E}[\mathbf{hd} (\mathbf{nil} t_H)]_H \rightarrow \mathcal{E}[\mathbf{wrong} t_H \text{ “Empty list”}]$

$\Gamma \vdash_H \mathbf{hd} (\mathbf{nil} t_H) : t'_H$  by the premise and lemma 2.  $\Gamma \vdash_H \mathbf{nil} t_H : \{t_H\}, t'_H = t_H$ , and  $\Gamma \vdash_H \mathbf{wrong} t_H \text{ “Empty list”} : t_H$  by lemmas 1 and 2.

**Case 4.9.**  $\mathcal{E}[\mathbf{tl} (\mathbf{nil} t_H)]_H \rightarrow \mathcal{E}[\mathbf{wrong} \{t_H\} \text{ “Empty list”}]$

$\Gamma \vdash_H \mathbf{tl} (\mathbf{nil} t_H) : t'_H$  by the premise and lemma 2.  $\Gamma \vdash_H \mathbf{nil} t_H : \{t_H\}, t'_H = \{t_H\}$ , and  $\Gamma \vdash_H \mathbf{wrong} \{t_H\} \text{ “Empty list”} : \{t_H\}$  by lemmas 1 and 2.

**Case 4.10.**  $\mathcal{E}[\mathbf{hd} (\mathbf{cons} e_H e'_H)]_H \rightarrow \mathcal{E}[e_H]$

$\Gamma \vdash_H \text{hd} (\text{cons } e_H e'_H) : t_H$  by the premise and lemma 2.  $\Gamma \vdash_H \text{cons } e_H e'_H : \{t_H\}$  and  $\Gamma \vdash_H e_H : t_H$  by lemmas 1 and 2.

**Case 4.11.**  $\mathcal{E}[\text{tl} (\text{cons } e_H e'_H)]_H \rightarrow \mathcal{E}[e'_H]$

$\Gamma \vdash_H \text{tl} (\text{cons } e_H e'_H) : t_H$  by the premise and lemma 2.  $t_H = \{t'_H\}$ ,  $\Gamma \vdash_H \text{cons } e_H e'_H : \{t'_H\}$ , and  $\Gamma \vdash_H e'_H : \{t'_H\}$  by lemmas 1 and 2.

**Case 4.12.**  $\mathcal{E}[\text{null?} (\text{nil } t_H)]_H \rightarrow \mathcal{E}[\bar{0}]$

$\Gamma \vdash_H \text{null?} (\text{nil } t_H) : t_H$  by the premise and lemma 2.  $t_H = \mathbf{N}$  and  $\vdash_H \bar{0} : \mathbf{N}$  by lemmas 1 and 2.

**Case 4.13.**  $\mathcal{E}[\text{null?} (\text{cons } e_H e'_H)]_H \rightarrow \mathcal{E}[\bar{1}]$

$\Gamma \vdash_H \text{null?} (\text{cons } e_H e'_H) : t_H$  by the premise and lemma 2.  $t_H = \mathbf{N}$  and  $\vdash_H \bar{1} : \mathbf{N}$  by lemmas 1 and 2.

**Case 4.14.**  $\mathcal{E}[\text{hm } t_H \text{ L } (\text{mh } \text{L } t'_H e_H)]_H \rightarrow \mathcal{E}[e_H]$  ( $t_H = t'_H$  and  $t_H \neq \text{L}$ )

*TEST*

**Case 4.15.**  $\mathcal{E}[\text{hm } t_H \text{ L } (\text{mh } \text{L } t'_H e_H)]_H \rightarrow \mathcal{E}[\text{wrong } t_H \text{ "Type mismatch"}]$  ( $t_H \neq t'_H$  and  $t_H \neq \text{L}$ )

*TEST*

**Case 4.16.**  ${}^L A B {}^L ({}^L B S v_S) \rightarrow {}^L A S v_S$  where  $(A, B) \in \{(H, M), (M, H)\}$

$\Gamma \vdash_A {}^L A B {}^L ({}^L B S v_S) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_S v_S : T S T$  by inversion (Lemma ??).  $\Gamma \vdash_B {}^L B S v_S : L$  and  $T = L$  by inversion and uniqueness of types.  $\Gamma \vdash_A {}^L A S v_S : L$  by inversion and uniqueness of types.  $\Gamma \vdash_A {}^L A S v_S : T$  because  $L = T$ .

**Case 4.17.**  ${}^N A B {}^N \bar{n} \rightarrow \bar{n}$  where  $(A, B) \in \{(H, M), (M, H)\}$

$\vdash_A {}^N AB^N \bar{n} : T$  by premise and uniqueness of types (Lemma ??).  $\vdash_A \bar{n} : N$  and  $T = N$  by inversion (Lemma ??) and uniqueness of types.

**Case 4.18.**  ${}^N AS \bar{n} \rightarrow \bar{n}$  where  $A \in \{H, M\}$

$\vdash_A {}^N AS \bar{n} : T$  by premise and uniqueness of types (Lemma ??).  $\vdash_A \bar{n} : N$  and  $T = N$  by inversion (Lemma ??) and uniqueness of types.

**Case 4.19.**  ${}^N AS v_S \rightarrow {}^N AS (\text{wrong "Not a number"}) (v_S \neq \bar{n})$  where  $A \in \{H, M\}$

$\Gamma \vdash_A {}^N AS v_S : T$  by premise and uniqueness of types (Lemma ??).  $T = N$  by inversion (Lemma ??) and uniqueness of types.  $\vdash_S \text{wrong "Not a number"} : TST$  by inversion.  $\vdash_A {}^N AS (\text{wrong "Not a number"}) : N$  by inversion and uniqueness of types.  $\vdash_A {}^N AS (\text{wrong "Not a number"}) : T$  because  $N = T$ .

**Case 4.20.**  ${}^{[T_1]} AB^{[T_1]} \text{nil}^{T_1} \rightarrow \text{nil}^{T_1}$  where  $(A, B) \in \{(H, M), (M, H)\}$

$\Gamma \vdash_A {}^{[T_1]} AB^{[T_1]} \text{nil}^{T_1} : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_A \text{nil}^{T_1} : [T_1]$  and  $T = [T_1]$  by inversion (Lemma ??) and uniqueness of types.  $\Gamma \vdash_A \text{nil}^{T_1} : T$  because  $[T_1] = T$ .

**Case 4.21.**  ${}^{[T_1]} AS \text{nil} \rightarrow \text{nil}^{T_1}$  where  $A \in \{H, M\}$

$\Gamma \vdash_A {}^{[T_1]} AS \text{nil} : T$  by premise and uniqueness of types (Lemma ??).  $T = [T_1]$  and  $\Gamma \vdash_A \text{nil}^{T_1} : [T_1]$  by inversion (Lemma ??) and uniqueness of types.  $\Gamma \vdash_A \text{nil}^{T_1} : T$  because  $[T_1] = T$ .

**Case 4.22.**  ${}^{[T_1]} HM^{[T_1]} (\text{cons } v_M^1 v_M^2) \rightarrow \text{cons } ({}^{T_1} HM^{T_1} v_M^1) ({}^{[T_1]} HM^{[T_1]} v_M^2)$

${}^{[T_1]} HM^{[T_1]} (\text{cons } v_M^1 v_M^2) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_M v_M^1 : T_1$ ,  $\Gamma \vdash_M v_M^2 : [T_1]$ ,  $\Gamma \vdash_M \text{cons } v_M^1 v_M^2 : [T_1]$ ,  $T = [T_1]$ ,  $\Gamma \vdash_H {}^{T_1} HM^{T_1} v_M^1 : T_1$ ,  $\Gamma \vdash_H {}^{[T_1]} HM^{[T_1]} v_M^2 : [T_1]$ , and  $\Gamma \vdash_H \text{cons } ({}^{T_1} HM^{T_1} v_M^1) ({}^{[T_1]} HM^{[T_1]} v_M^2) : T$ .



$v_M^1$ )  $(^{[T_1]}HM^{[T_1]} v_M^2) : [T_1]$  by inversion (Lemma ??) and uniqueness of types.

$\Gamma \vdash_H \text{cons } (^{T_1}HM^{T_1} v_M^1) (^{[T_1]}HM^{[T_1]} v_M^2) : T$  because  $[T_1] = T$ .

**Case 4.23.**  $^{[T_1]}AS (\text{cons } v_S^1 v_S^2) \rightarrow \text{cons } (^{T_1}AS v_S^1) (^{[T_1]}AS v_S^2)$  where  $A \in \{H, M\}$

$\Gamma \vdash_A ^{[T_1]}AS (\text{cons } v_S^1 v_S^2) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_S v_S^1 : TST$ ,  $\Gamma \vdash_S v_S^2 : TST$ , and  $\Gamma \vdash_S \text{cons } v_S^1 v_S^2 : TST$  by inversion (Lemma ??).  $T = [T_1]$ ,  $\Gamma \vdash_A ^{T_1}AS v_S^1 : T_1$ ,  $\Gamma \vdash_A ^{[T_1]}AS v_S^2 : [T_1]$ , and  $\Gamma \vdash_A \text{cons } (^{T_1}AS v_S^1) (^{[T_1]}AS v_S^2) : [T_1]$  by inversion and uniqueness of types.  $\Gamma \vdash_A \text{cons } (^{T_1}AS v_S^1) (^{[T_1]}AS v_S^2) : T$  because  $[T_1] = T$ .

**Case 4.24.**  $^{[T_1]}HM^{[T_1]} (^{[T_1]}MH^{[T_1]} (\text{cons } e_H^1 e_H^2)) \rightarrow \text{cons } e_H^1 e_H^2$

$^{[T_1]}HM^{[T_1]} (^{[T_1]}MH^{[T_1]} (\text{cons } e_H^1 e_H^2)) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_H \text{cons } e_H^1 e_H^2 : [T_1]$ ,  $\Gamma \vdash_H ^{[T_1]}MH^{[T_1]} (\text{cons } e_H^1 e_H^2) : [T_1]$ , and  $T = [T_1]$  by inversion (Lemma ??) and uniqueness of types.  $\Gamma \vdash_H \text{cons } e_H^1 e_H^2 : T$  because  $[T_1] = T$ .

**Case 4.25.**  $^{[T_1]}HS (SH^{[T_1]} (\text{cons } e_H^1 e_H^2)) \rightarrow \text{cons } e_H^1 e_H^2$

$^{[T_1]}HS (SH^{[T_1]} (\text{cons } e_H^1 e_H^2)) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_H \text{cons } e_H^1 e_H^2 : [T_1]$  by inversion (Lemma ??) and uniqueness of types.  $\Gamma \vdash_S SH^{[T_1]} (\text{cons } e_H^1 e_H^2) : TST$  by inversion.  $T = [T_1]$  by inversion and uniqueness of types.  $\Gamma \vdash_H \text{cons } e_H^1 e_H^2 : T$  because  $[T_1] = T$ .

**Case 4.26.**  $^{[T_1]}AS v_S^1 \rightarrow ^{[T_1]}AS (\text{wrong "Not a list"}) (v_S^1 \neq \text{cons } v_S^2 v_S^3 \text{ and } v_S^1 \neq \text{nil})$  where  $A \in \{H, M\}$

$\Gamma \vdash_A ^{[T_1]}AS v_S^1 : T$  by premise and uniqueness of types (Lemma ??).  $T = [T_1]$  by inversion (Lemma ??) and uniqueness of types.  $\vdash_S \text{wrong "Not a list"} : TST$  by inversion.  $\Gamma \vdash_A ^{[T_1]}AS (\text{wrong "Not a list"}) : [T_1]$  by inversion and uniqueness

of types.  $\Gamma \vdash_A {}^{[T_1]}AS$  (**wrong** “Not a list”) :  $T$  because  $[T_1] = T$ .

**Case 4.27.**  $T_1^a AS (SA^{T_1^a} B_A) \rightarrow B_A$  where  $(A, B) \in \{(H, e), (M, v)\}$

$T_1^a AS (SA^{T_1^a} B_A) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_A B_A : T_1^a[T_i/T_i^a]$  by inversion (Lemma ??) and uniqueness of types.  $\Gamma \vdash_S SA^{T_1^a} B_A : TST$  by inversion.  $T = T_1^a[T_i/T_i^a]$  by inversion and uniqueness of types.  $\Gamma \vdash_A B_A : T$  because  $T_1^a[T_i/T_i^a] = T$ .

**Case 4.28.**  $T_1^a AS v_S \rightarrow T_1^a AS$  (**wrong** “Parametricity violated”) ( $v_S \neq SA^{T_1^a} B_A$ ) where  $(A, B) \in \{(H, e), (M, v)\}$

$\Gamma \vdash_A T_1^a AS v_S : T$  by premise and uniqueness of types (Lemma ??).  $T = T_1^a[T_i/T_i^a]$  by inversion (Lemma ??) and uniqueness of types.  $\vdash_S$  **wrong** “Parametricity violated” :  $TST$  by inversion.  $T_1^a AS$  (**wrong** “Parametricity violated”) :  $T_1^a[T_i/T_i^a]$  by inversion and uniqueness of types.  $T_1^a AS$  (**wrong** “Parametricity violated”) :  $T$  because  $T_1^a[T_i/T_i^a] = T$ .

**Case 4.29.**  $\mathcal{E}[\text{hm } (t_H \rightarrow t'_H) (t_M \rightarrow t'_M) (\lambda x_M : t''_M.e_M)]_H \rightarrow \mathcal{E}[\lambda x_H : t_H.\text{hm } t'_H t'_M ((\lambda x_M : t''_M.e_M) (\text{mh } t_M t_H x_H))]$

$T_1 \rightarrow T_2 AB^{T_1 \rightarrow T_2} (\lambda x_1 : T_1.e_B) \rightarrow \lambda x_2 : T_1.(T_2 AB^{T_2} ((\lambda x_1 : T_1.e_B) (T_1 BA^{T_1} x_2)))$   
where  $(A, B) \in \{(H, M), (M, H)\}$

$\Gamma \vdash_A T_1 \rightarrow T_2 AB^{T_1 \rightarrow T_2} (\lambda x_1 : T_1.e_B) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_B \lambda x_1 : T_1.e_B : T_1 \rightarrow T_2$ ,  $T = T_1 \rightarrow T_2$ ,  $\Gamma, x_2 : T_1 \vdash_A x_2 : T_1$ ,  $\Gamma, x_2 : T_1 \vdash_B T_1 BA^{T_1} x_2 : T_1$ ,  $\Gamma, x_2 : T_1 \vdash_B (\lambda x_1 : T_1.e_B) (T_1 BA^{T_1} x_2) : T_2$ ,  $\Gamma, x_2 : T_1 \vdash_A T_2 AB^{T_2} ((\lambda x_1 : T_1.e_B) (T_1 BA^{T_1} x_2)) : T_2$ , and  $\Gamma \vdash_A \lambda x_2 : T_1.(T_2 AB^{T_2} ((\lambda x_1 : T_1.e_B) (T_1 BA^{T_1} x_2))) : T_1 \rightarrow T_2$  by inversion (Lemma ??) and uniqueness of types.  $\Gamma \vdash_A \lambda x_2 : T_1.(T_2 AB^{T_2} ((\lambda x_1 : T_1.e_B) (T_1 BA^{T_1} x_2))) : T$  because  $T_1 \rightarrow T_2 = T$ .

**Case 4.30.**  $T_1 \rightarrow T_2 AS (\lambda x_1. e_S) \rightarrow \lambda x_2 : T_1[T_i/T_i^a].(T_2 AS ((\lambda x_1. e_S) (SA^{T_1} x_2)))$   
where  $A \in \{H, M\}$

$\Gamma \vdash_A T_1 \rightarrow T_2 AS (\lambda x_1. e_S) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_S \lambda x_1. e_S : TST$  by inversion (Lemma ??).  $T = (T_1 \rightarrow T_2)[T_i/T_i^a]$  by inversion and uniqueness of types.  $\Gamma, x_2 : T_1[T_i/T_i^a] \vdash_A x_2 : T_1[T_i/T_i^a]$  by inversion and uniqueness of types.  $\Gamma, x_2 : T_1[T_i/T_i^a] \vdash_S SA^{T_1} x_2 : TST$  and  $\Gamma, x_2 : T_1[T_i/T_i^a] \vdash_S (\lambda x_1. e_S) (SA^{T_1} x_2) : TST$  by inversion.  $\Gamma, x_2 : T_1[T_i/T_i^a] \vdash_A T_2 AS ((\lambda x_1. e_S) (SA^{T_1} x_2)) : T_2[T_i/T_i^a]$  and  $\Gamma \vdash_A \lambda x_2 : T_1[T_i/T_i^a].(T_2 AS ((\lambda x_1. e_S) (SA^{T_1} x_2))) : T_1[T_i/T_i^a] \rightarrow T_2[T_i/T_i^a]$  by inversion and uniqueness of types.  $\Gamma \vdash_A \lambda x_2 : T_1[T_i/T_i^a].(T_2 AS ((\lambda x_1. e_S) (SA^{T_1} x_2))) : T$  because  $T_1[T_i/T_i^a] \rightarrow T_2[T_i/T_i^a] = (T_1 \rightarrow T_2)[T_i/T_i^a] = T$ .

**Case 4.31.**  $T_1 \rightarrow T_2 AS v_S \rightarrow T_1 \rightarrow T_2 AS (\text{wrong "Not a function"}) (v_S \neq \lambda x. e_S)$   
where  $A \in \{H, M\}$

$\Gamma \vdash_A T_1 \rightarrow T_2 AS v_S : T$  by premise and uniqueness of types (Lemma ??).  $T = (T_1 \rightarrow T_2)[T_i/T_i^a]$  by inversion (Lemma ??) and uniqueness of types.  $\vdash_S \text{wrong "Not a function"} : TST$  by inversion.  $\Gamma \vdash_A T_1 \rightarrow T_2 AS (\text{wrong "Not a function"}) : (T_1 \rightarrow T_2)[T_i/T_i^a]$  by inversion and uniqueness of types.  $\Gamma \vdash_A T_1 \rightarrow T_2 AS (\text{wrong "Not a function"}) : T$  because  $(T_1 \rightarrow T_2)[T_i/T_i^a] = T$ .

**Case 4.32.**  $\forall X. T_1 AB^{\forall X. T_1} (\Lambda X. e_B) \rightarrow \Lambda X. (T_1 AB^{T_1} e_B)$  where  $(A, B) \in \{(H, M), (M, H)\}$

$\Gamma \vdash_A \forall X. T_1 AB^{\forall X. T_1} (\Lambda X. e_B) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma, X \vdash_B e_B : T_1$ ,  $\Gamma \vdash_B \Lambda X. e_B : \forall X. T_1$ ,  $T = \forall X. T_1$ ,  $\Gamma, X \vdash_A T_1 AB^{T_1} e_B : T_1$ , and  $\Gamma \vdash_A \Lambda X. (T_1 AB^{T_1} e_B) : \forall X. T_1$  by inversion (Lemma ??) and uniqueness of types.  $\Gamma \vdash_A \Lambda X. (T_1 AB^{T_1} e_B) : T$  because  $\forall X. T_1 = T$ .

**Case 4.33.**  $\forall X. T_1 AB^{\forall X. T_1} (\forall X. T_1 BS v_S) \rightarrow \forall X. T_1 AS v_S$  where  $(A, B) \in \{(H, M),$

$(M, H)\}$

$\Gamma \vdash_A \forall X.T_1 AB^{\forall X.T_1} (\forall X.T_1 BS \ v_S) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_S v_S : TST$  by inversion (Lemma ??).  $\Gamma \vdash_B \forall X.T_1 BS \ v_S : \forall X.T_1$ ,  $T = \forall X.T_1$ , and  $\Gamma \vdash_A \forall X.T_1 AS \ v_S : \forall X.T_1$  by inversion and uniqueness of types.  $\Gamma \vdash_A \forall X.T_1 AS \ v_S : T$  because  $\forall X.T_1 = T$ .

**Case 4.34.**  $(\forall X.T_1 AS \ v_S) \{T_2\} \rightarrow T_1[T_2^a/X] AS \ v_S$  where  $A \in \{H, M\}$

$\Gamma \vdash_A (\forall X.T_1 AS \ v_S) \{T_2\} : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_S v_S : TST$  by inversion (Lemma ??).  $\Gamma \vdash_A \forall X.T_1 AS \ v_S : \forall X.T_1$  and  $T = T_1[T_2/X]$  by inversion and uniqueness of types.  $\Gamma \vdash_A T_1[T_2^a/X] AS \ v_S : T_1[T_2^a/X][T_i/T_i^a]$  by inversion and uniqueness of types.  $\Gamma \vdash_A T_1[T_2^a/X] AS \ v_S : T$  because  $T_1[T_2^a/X][T_i/T_i^a] = T_1[T_2/X] = T$ .

□

### Theorem 5. ML Preservation

If  $\Gamma \vdash_M e_M : t_M$  and  $e_M \rightarrow e'_M$  then  $\Gamma \vdash_M e'_M : t_M$ .

*Proof.* By cases on the reductions  $e_A^1 \rightarrow e_A^2$  and  $e_S^1 \rightarrow e_S^2$  and evaluation context preservation (Lemma ??). Straightforward cases of Scheme preservation are elided.

**Case 5.1.**  $\text{hd} (\text{cons } v_M^1 \ v_M^2) \rightarrow v_M^1$

$\Gamma \vdash_M \text{hd} (\text{cons } v_M^1 \ v_M^2) : T$  by premise and uniqueness of types (Lemma ??).  $\Gamma \vdash_M v_M^1 : T_1$ ,  $\Gamma \vdash_M \text{cons } v_M^1 \ v_M^2 : [T_1]$ , and  $T = T_1$  by inversion (Lemma ??) and uniqueness of types.  $\Gamma \vdash_M v_M^1 : T$  because  $T_1 = T$ .

**Case 5.2.**  $\text{tl} (\text{cons } v_M^1 \ v_M^2) \rightarrow v_M^2$

$\Gamma \vdash_M \mathbf{tl} (\mathbf{cons} v_M^1 v_M^2) : T$  by premise and uniqueness of types (Lemma ??).  
 $\Gamma \vdash_M v_M^2 : [T_1]$ ,  $\Gamma \vdash_M \mathbf{cons} v_M^1 v_M^2 : [T_1]$ , and  $T = [T_1]$  by inversion (Lemma ??)  
and uniqueness of types.  $\Gamma \vdash_M v_M^2 : T$  because  $[T_1] = T$ .

**Case 5.3.**  $\mathbf{hd} ([T_1]MH^{[T_1]} (\mathbf{cons} e_H^1 e_H^2)) \rightarrow T_1 MH^{T_1} e_H^1$

$\Gamma \vdash_M \mathbf{hd} ([T_1]MH^{[T_1]} (\mathbf{cons} e_H^1 e_H^2)) : T$  by premise and uniqueness of types  
(Lemma ??).  $\Gamma \vdash_H e_H^1 : T_1$ ,  $\Gamma \vdash_H \mathbf{cons} e_H^1 e_H^2 : [T_1]$ ,  $\Gamma \vdash_M [T_1]MH^{[T_1]} (\mathbf{cons} e_H^1$   
 $e_H^2) : [T_1]$ ,  $T = T_1$ , and  $T_1 MH^{T_1} e_H^1 : T_1$  by inversion (Lemma ??) and uniqueness  
of types.  $T_1 MH^{T_1} e_H^1 : T$  because  $T_1 = T$ .

**Case 5.4.**  $\mathbf{tl} ([T_1]MH^{[T_1]} (\mathbf{cons} e_H^1 e_H^2)) \rightarrow [T_1]MH^{[T_1]} e_H^2$

$\Gamma \vdash_M \mathbf{tl} ([T_1]MH^{[T_1]} (\mathbf{cons} e_H^1 e_H^2)) : T$  by premise and uniqueness of types  
(Lemma ??).  $\Gamma \vdash_H e_H^2 : [T_1]$ ,  $\Gamma \vdash_H \mathbf{cons} e_H^1 e_H^2 : [T_1]$ ,  $\Gamma \vdash_M [T_1]MH^{[T_1]} (\mathbf{cons}$   
 $e_H^1 e_H^2) : [T_1]$ ,  $T = [T_1]$ , and  $[T_1]MH^{[T_1]} e_H^2 : [T_1]$  by inversion (Lemma ??) and  
uniqueness of types.  $[T_1]MH^{[T_1]} e_H^2 : T$  because  $[T_1] = T$ .

**Case 5.5.**  $\mathbf{null?} ([T]MH^{[T]} (\mathbf{cons} e_H^1 e_H^2)) \rightarrow \bar{1}$

$\Gamma \vdash_M \mathbf{null?} ([T]MH^{[T]} (\mathbf{cons} e_H^1 e_H^2)) : T$  by premise and uniqueness of types  
(Lemma ??).  $T = N$  and  $\vdash_A \bar{1} : N$  by inversion (Lemma ??) and uniqueness of  
types.  $\vdash_A \bar{1} : T$  because  $N = T$ .

**Case 5.6.**  $[T_1]MS (SH^{[T_1]} (\mathbf{cons} e_H^1 e_H^2)) \rightarrow [T_1]MH^{[T_1]} (\mathbf{cons} e_H^1 e_H^2)$

$[T_1]MS (SH^{[T_1]} (\mathbf{cons} e_H^1 e_H^2)) : T$  by premise and uniqueness of types (Lemma  
??).  $\Gamma \vdash_H \mathbf{cons} e_H^1 e_H^2 : [T_1]$  by inversion (Lemma ??) and uniqueness of types.  
 $\Gamma \vdash_S SH^{[T_1]} (\mathbf{cons} e_H^1 e_H^2) : TST$  by inversion.  $T = [T_1]$  and  $\Gamma \vdash_M [T_1]MH^{[T_1]}$   
 $(\mathbf{cons} e_H^1 e_H^2) : [T_1]$  by inversion and uniqueness of types.  $\Gamma \vdash_M [T_1]MH^{[T_1]} (\mathbf{cons}$   
 $e_H^1 e_H^2) : T$  because  $[T_1] = T$ .

□

**Theorem 6.** *Scheme Preservation*

If  $\Gamma \vdash_S e_S : \text{TST}$  and  $e_S \rightarrow e'_S$  then  $\Gamma \vdash_S e'_S : \text{TST}$ .

*Proof.* By cases on the reductions  $e_A^1 \rightarrow e_A^2$  and  $e_S^1 \rightarrow e_S^2$  and evaluation context preservation (Lemma ??). Straightforward cases of Scheme preservation are elided.

**Case 6.1.**  $\text{hd} (SH^{[T_1]} (\text{cons } e_H^1 e_H^2)) \rightarrow SH^{T_1} e_H^1$

$\Gamma \vdash_S \text{hd} (SH^{[T_1]} (\text{cons } e_H^1 e_H^2)) : \text{TST}$  by premise.  $\Gamma \vdash_H e_H^1 : T_1$  and  $\Gamma \vdash_H \text{cons } e_H^1 e_H^2 : [T_1]$  by inversion (Lemma ??) and uniqueness of types (Lemma ??).  $\Gamma \vdash_S SH^{[T_1]} (\text{cons } e_H^1 e_H^2) : \text{TST}$  and  $\Gamma \vdash_S SH^{T_1} e_H^1 : \text{TST}$  by inversion.

**Case 6.2.**  $\text{tl} (SH^{[T_1]} (\text{cons } e_H^1 e_H^2)) \rightarrow SH^{[T_1]} e_H^2$

$\Gamma \vdash_S \text{tl} (SH^{[T_1]} (\text{cons } e_H^1 e_H^2)) : \text{TST}$  by premise.  $\Gamma \vdash_H e_H^2 : [T_1]$  and  $\Gamma \vdash_H \text{cons } e_H^1 e_H^2 : [T_1]$  by inversion (Lemma ??) and uniqueness of types (Lemma ??).  $\Gamma \vdash_S SH^{[T_1]} (\text{cons } e_H^1 e_H^2) : \text{TST}$  and  $\Gamma \vdash_S SH^{[T_1]} e_H^2 : \text{TST}$  by inversion.

**Case 6.3.**  $SM^{[T_1]} (\text{cons } v_M^1 v_M^2) \rightarrow \text{cons} (SM^{T_1} v_M^1) (SM^{[T_1]} v_M^2)$

$\Gamma \vdash_S SM^{[T_1]} (\text{cons } v_M^1 v_M^2) : \text{TST}$  by premise.  $\Gamma \vdash_M v_M^1 : T_1$ ,  $\Gamma \vdash_M v_M^2 : [T_1]$ , and  $\Gamma \vdash_M \text{cons } v_M^1 v_M^2 : [T_1]$  by inversion (Lemma ??) and uniqueness of types (Lemma ??).  $\Gamma \vdash_S SM^{T_1} v_M^1 : \text{TST}$ ,  $\Gamma \vdash_S SM^{[T_1]} v_M^2 : \text{TST}$ , and  $\Gamma \vdash_S \text{cons} (SM^{T_1} v_M^1) (SM^{[T_1]} v_M^2) : \text{TST}$  by inversion.

**Case 6.4.**  $SM^{[T_1]} ([T_1]MH^{[T_1]} (\text{cons } e_H^1 e_H^2)) \rightarrow SH^{[T_1]} (\text{cons } e_H^1 e_H^2)$

$SM^{[T_1]} ([T_1]MH^{[T_1]} (\text{cons } e_H^1 e_H^2)) : \text{TST}$  by premise.  $\Gamma \vdash_H \text{cons } e_H^1 e_H^2 : [T_1]$  and  $\Gamma \vdash_M [T_1]MH^{[T_1]} (\text{cons } e_H^1 e_H^2) : [T_1]$  by inversion (Lemma ??) and uniqueness of types (Lemma ??).  $\Gamma \vdash_S SH^{[T_1]} (\text{cons } e_H^1 e_H^2) : \text{TST}$  by inversion.

**Case 6.5.**  $SA^{T_1 \rightarrow T_2} (\lambda x_1 : T_1[T_i/T_i^a].e_A) \rightarrow \lambda x_2.(SA^{T_2} ((\lambda x_1 : T_1[T_i/T_i^a].e_A) (T_1 AS x_2))))$  where  $A \in \{H, M\}$

$\Gamma \vdash_S SA^{T_1 \rightarrow T_2} (\lambda x_1 : T_1[T_i/T_i^a].e_A) : TST$  by premise.  $\Gamma \vdash_A \lambda x_1 : T_1[T_i/T_i^a].e_A : T_1[T_i/T_i^a] \rightarrow T_2[T_i/T_i^a]$  by inversion (Lemma ??) and uniqueness of types (Lemma ??).  $\Gamma, x_2 : TST \vdash_S x_2 : TST$  by inversion.  $\Gamma, x_2 : TST \vdash_A T_1 AS x_2 : T_1[T_i/T_i^a]$  and  $\Gamma, x_2 : TST \vdash_A (\lambda x_1 : T_1[T_i/T_i^a].e_A) (T_1 AS x_2) : T_2[T_i/T_i^a]$  by inversion and uniqueness of types.  $\Gamma, x_2 : TST \vdash_S SA^{T_2} ((\lambda x_1 : T_1[T_i/T_i^a].e_A) (T_1 AS x_2)) : TST$  and  $\Gamma \vdash_S \lambda x_2.(SA^{T_2} ((\lambda x_1 : T_1[T_i/T_i^a].e_A) (T_1 AS x_2))) : TST$  by inversion.

**Case 6.6.**  $SA^{\forall X.T_1} (\Lambda X.e_A) \rightarrow SA^{T_1[L/X]} ((\Lambda X.e_A) \{L\})$  where  $A \in \{H, M\}$

$\Gamma \vdash_S SA^{\forall X.T_1} (\Lambda X.e_A) : TST$  by premise.  $\Gamma \vdash_A \Lambda X.e_A : \forall X.T_1$  and  $\Gamma \vdash_A (\Lambda X.e_A) \{L\} : T_1[L/X]$  by inversion (Lemma ??) and uniqueness of types (Lemma ??).  $\Gamma \vdash_S SA^{T_1[L/X]} ((\Lambda X.e_A) \{L\}) : TST$  by inversion.

□

# Chapter 4

## Related Work

This work extends the work of Kinghorn [?] by adding Haskell and lists to his model of computation, his proof of type soundness, and his implementation of the model. Kinghorn extended the work of Matthews and Findler [?] by adding parametric polymorphism and parametricity to their model of computation, providing a more thorough proof of its type soundness, and implementing it with a fully-featured Scheme and a subset of Objective Caml, a dialect of ML.

Guha et al. [?] describe a system of parametric polymorphic contracts for higher-order functions that assign blame for contract violations and protect parametricity. The system both ensures function arguments match the contract parameters and prevents functions from examining the types and values of their arguments. This work uses two separate mechanisms, boundary expressions and label types, to achieve the same result.

Perhaps the most mainstream systems of interoperation are the Common Object Request Broker Architecture (CORBA), the Component Object Model (COM), and the .NET Framework, yet not one of them supports interoperation



between Haskell, ML, and Scheme as this work does. CORBA, COM, and the .NET Framework support the interoperation of static and dynamic type systems and strict evaluation, but not higher-order functions, parametric polymorphism, parametricity, or lazy evaluation [?] [?] [?].

Tobin-Hochstadt and Felleisen [?] describe a system of mechanically translating programs written in a dynamically-typed language to an equivalent form in a similar, statically-typed language. The system has higher-order functions, static and dynamic type systems, and strict evaluation, but not parametric polymorphism, parametricity, or lazy evaluation. The system enables the interoperation of higher-order functions, dynamic type systems, and strict evaluation, but not parametric polymorphism, parametricity, static type systems, or lazy evaluation. It uses contracts for higher-order functions to assign blame to languages for type errors, which this model does not do.

Henglein and Rehof [?] describe a system of polymorphic type inference for Scheme that infers types and run-time type operations, thereby giving a high-level translation from Scheme to ML. ML programs cannot be translated to equivalent Scheme programs. The system has higher-order functions, parametric polymorphism, parametricity, static and dynamic type systems, and strict evaluation, but not lazy evaluation. The system enables the interoperation of higher-order functions, dynamic type systems, and strict evaluation, but not parametric polymorphism, parametricity, static type systems, or lazy evaluation.

Benton [?] describes a system of embedding dynamically-typed languages within the statically-typed language ML and projecting dynamically-typed values back into ML. The system has higher-order functions, parametric polymorphism, parametricity, static and dynamic type systems, and strict evaluation, but does not have lazy evaluation. The system enables the interoperation of higher-order

functions, parametric polymorphism, static and dynamic type systems, and strict evaluation, but not parametricity or lazy evaluation.

# Chapter 5

## Future Work

The model of computation is sufficient to enable the interoperation of languages with incompatible evaluation strategies. Certainly other data types could be added to the model, but they would add nothing new to the method of resolving incompatible evaluation strategies and would further complicate the model. The implementation of the model would be more useful if additional language constructs and data types were added. Performance would improve if modules were compiled to bytecodes or machine code. Adding languages with other evaluation strategies, such as normal order and applicative order, or languages with static type systems that do not support parametricity, would be interesting, but the sizes of the model and proof would grow exponentially. Further explorations of incompatible evaluation strategies would best be tackled with pairs of languages to minimize complexity.

# Chapter 6

## Conclusion

This work resolved three language incompatibilities in a system of interoperation for three diverse languages. It resolved incompatible type systems with contracts for higher-order functions and lump types. It resolved incompatible support for parametricity with label types. It resolved incompatible evaluation strategies with delayed conversions for list constructions. It defined a model of computation that can express interoperation where the aforementioned incompatibilities arise and resolve them, provided a proof of its type soundness, and described an implementation of it that supported additional language features.