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 Faught

June 2008

AUTHORIZATION FOR REPRODUCTION OF MASTER'S THESIS

Signature

Date

APPROVAL PAGE

TITLE: Interoperation for Incompatible Evaluation Strategies

AUTHOR: William Faught

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 Faught

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

List of Figures **vii**

1 Introduction **1**

2 Model of Computation **4**

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

3 Proof of Type Soundness **34**

3.1 Proof of Expression Progress 34

3.2 Proof of Type Preservation 45

4 Related Work **57**

5 Future Work **60**

6 Conclusion **61**

Bibliography **62**

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. Adhoc polymorphism in dynamically-typed languages enables argument types to determine polymorphic function behavior. Since determining function behavior is undecidable [2], actual types for these functions cannot be reliably calculated at language boundaries and compared to expected types. Instead, they are wrapped in contracts [4] 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 [7], which extended the model of Matthews and Findler [8].

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 $T \to T$, and foralls $\forall X.T$. Haskell values v_H comprise functions $\lambda x : T.e_H$, type abstractions $\Lambda X.e_H$, natural numbers $\overline{n}$, empty lists $\texttt{nil}^T$, list constructions $\texttt{cons}\ e_H\ e_H$, Scheme boundaries with lump expected types ${}^{L}HS\ v_S$, 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 $\texttt{null?}\ e_H$, conditions $\texttt{if0}\ e_H\ e_H\ e_H$, list operations $\texttt{hd}\ e_H$ and $\texttt{tl}\ e_H$, fixed-point operations $\texttt{fix}\ e_H$, error reports $\texttt{wrong}^T$ string, ML boundaries ${}^{T}HM^{T}\ e_M$, and Scheme boundaries ${}^{T}HS\ e_S$. 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.

$\overline{n}$ 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 ${}^{T}MH^{T}\ e_H$. ML forced values comprise functions, type abstractions, natural numbers, empty lists, list constructions $\texttt{cons}\ v_M\ v_M$, Scheme boundaries with lump expected types ${}^{L}MS\ v_S$, Scheme boundaries with forall expected types ${}^{\forall X.T}MS\ v_S$, and Haskell boundaries with list expected types ${}^{[T]}MH^{[T]}\ (\texttt{cons}\ e_H\ e_H)$. ML expressions comprise unforced values u_M, variables, function applications, type applications, arithmetic operations, empty list predicates, conditions, list constructions $\texttt{cons}\ e_M\ e_M$, list operations, fixed-point operations, error reports, and ML boundaries ${}^{T}MS\ e_S$. 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 $SH^{T}\ e_H$. Scheme forced values comprise functions, natural numbers, empty lists, list constructions $\texttt{cons}\ v_S\ v_S$, Haskell boundaries with list actual types $SH^{[T]}\ (\texttt{cons}\ e_H\ e_H)$, Haskell boundaries with label actual types $SH^{T^a}\ v_H$, and ML boundaries with label actual types $SM^{T^a}\ v_M$. 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 $\texttt{fun?}\ e_S$, lists $\texttt{list?}\ e_S$, and natural numbers $\texttt{nat?}\ e_S$. 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 Γ 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 Γ. 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 unambigious.

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

2.3 Operational Semantics

$e^1_H[e^2_H/x]$ denotes the substitution of expression e^2_H for free occurrences of variable x within expression e^1_H. 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 $\mathscr{E}$. The evaluation of a single language instantiates $\mathscr{E}$ to E_H for Haskell, to E_M for ML, and to E_S for Scheme. Language interoperation instantiates $\mathscr{E}$ according to the language in which programs begin and end. $\mathscr{E}$ 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 (${}^{T}HM^{T}$ 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, ${}^{N}HM^{N}\ \overline{n}$ reduces to $\overline{n}$.

2.4.2 List Types

If a boundary has expected and actual list types (${}^{[T]}HM^{[T]}\ v_M$ for example), the value is either an empty list ($\texttt{nil}^T$ for example), a list construction ($\texttt{cons}\ v_M^1$ v_M^2 for example), or a Haskell list construction embedded in ML (${}^{[T]}MH^{[T]}$ ($\texttt{cons}$ $e_H^1\ e_H^2$) for example). If it is an empty list, the boundary reduces to the empty list. For example, ${}^{[T]}HM^{[T]}\ \texttt{nil}^T$ reduces to $\texttt{nil}^T$. 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, ${}^{[T]}HM^{[T]}$ (`cons` v_M^1 v_M^2) reduces to `cons` (${}^{T}HM^{T}$ 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 ${}^{[T]}MH^{[T]}$ (`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, `hd` (${}^{[T]}MH^{[T]}$ (`cons` e_H^1 e_H^2)) reduces to ${}^{T}MH^{T}$ e_H^1. If it is a Haskell list construction embedded in ML, the boundary reduces to the list construction. For example, ${}^{[T]}HM^{[T]}$ (${}^{[T]}MH^{[T]}$ (`cons` e_H^1 e_H^2)) reduces to `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 (${}^{[T]}MH^{[T]}$ (`cons` e_H^1 e_H^2) for example). If it is an empty list, the boundary reduces to an empty list. For example, $SH^{[T]}$ `nil`^{T} reduces to `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, $SM^{[T]}$ (`cons` v_M^1 v_M^2) reduces to `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 ${}^{[T]}MH^{[T]}$ (`cons` e_H^1 e_H^2) is irreducible, as discussed above. Therefore $SH^{[T]}$ (`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, `hd` ($SH^{[T]}$ (`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, $SM^{[T]}$ (${}^{[T]}MH^{[T]}$ (`cons` e_H^1 e_H^2)) reduces to $SH^{[T]}$ (`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 ($SH^{[T]}$ (`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, ${}^{[T]}HS$ `nil` reduces to $\texttt{nil}^T$. If it is a list construction, the boundary reduces to a list construction of the old head and tail wrapped in boundaries. For example, ${}^{[T]}HS$ (`cons` v_S^1 v_S^2) reduces to `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, ${}^{[T]}HS$ ($SH^{[T]}$ (`cons` e_H^1 e_H^2)) reduces to `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, ${}^{[T]}MS$ ($SH^{[T]}$ (`cons` e_H^1 e_H^2)) reduces to ${}^{[T]}MH^{[T]}$ (`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

$$
\begin{array}{rl}
& ({}^{N\to[N]}HS\ \lambda x_1.\mathtt{nil})\ \overline{0} \\
\to & (\lambda x_2 : N.({}^{[N]}HS\ ((\lambda x_1.\mathtt{nil})\ (SH^N\ x_2))))\ \overline{0} \\
\to & {}^{[N]}HS\ ((\lambda x_1.\mathtt{nil})\ (SH^N\ \overline{0})) \\
\to & {}^{[N]}HS\ ((\lambda x_1.\mathtt{nil})\ \overline{0}) \\
\to & {}^{[N]}HS\ \mathtt{nil} \\
\to & \mathtt{nil}^N
\end{array}
$$

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 $\overline{0}$ instead of `nil`, the computation would have reduced to ${}^{[N]}HS\ \overline{0}$ instead of ${}^{[N]}HS$ `nil`. Since $\overline{0}$ is not a list, ${}^{[N]}HS$ $\overline{0}$ 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 $\Lambda X.({}^{T}HM^{T}\ e_M)$. 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 (${}^{\forall X.T}HS$

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

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

$$
\begin{array}{ll}
 & ((^{\forall X.(X \to X)}HS\ \lambda x_1.(\texttt{if0}\ x_1\ \overline{1}\ x_1))\ \{N\})\ \overline{0} \\
\to & (^{N^a \to N^a}HS\ \lambda x_1.(\texttt{if0}\ x_1\ \overline{1}\ x_1))\ \overline{0} \\
\to & (\lambda x_2 : N.(^{N^a}HS\ ((\lambda x_1.(\texttt{if0}\ x_1\ \overline{1}\ x_1))\ (SH^{N^a}\ x_2))))\ \overline{0} \\
\to & ^{N^a}HS\ ((\lambda x_1.(\texttt{if0}\ x_1\ \overline{1}\ x_1))\ (SH^{N^a}\ \overline{0})) \\
\to & ^{N^a}HS\ (\texttt{if0}\ (SH^{N^a}\ \overline{0})\ \overline{1}\ (SH^{N^a}\ \overline{0})) \\
\to & ^{N^a}HS\ (SH^{N^a}\ \overline{0}) \\
\to & \overline{0}
\end{array}
$$

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 $\forall X.(X \to X)$, instantiated with N, to be $N \to N$. Observe that the application of $^{\forall X.(X \to X)}HS\ \lambda x_1.(\texttt{if0}\ x_1\ \overline{1}\ x_1)$, which has type $\forall X.(X \to X)$, to N reduces to $^{N^a \to N^a}HS\ \lambda x_1.(\texttt{if0}\ x_1\ \overline{1}\ x_1)$, which appears to have type $N^a \to N^a$. The Haskell and ML typing relations resolve this conflict by removing all labels from expected and actual types before making typing judgements. Therefore $^{N^a \to N^a}HS\ \lambda x_1.(\texttt{if0}\ x_1\ \overline{1}\ x_1)$ has type $N \to N$, 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.

$$
\begin{array}{ll}
& ((((^{\forall X_1.(\forall X_2.(X_1 \rightarrow (X_2 \rightarrow X_2)))} HS\ \lambda x_1.(\lambda x_2.x_1))\ \{N\})\ \{N\})\ \overline{0})\ \overline{1} \\
\rightarrow & (((^{\forall X_2.(N^a \rightarrow (X_2 \rightarrow X_2))} HS\ \lambda x_1.(\lambda x_2.x_1))\ \{N\})\ \overline{0})\ \overline{1} \\
\rightarrow & ((^{N^a \rightarrow (N^b \rightarrow N^b)} HS\ \lambda x_1.(\lambda x_2.x_1))\ \overline{0})\ \overline{1} \\
\rightarrow & ((\lambda x_3 : N.(^{N^b \rightarrow N^b} HS\ ((\lambda x_1.(\lambda x_2.x_1))\ (SH^{N^a}\ x_3))))\ \overline{0})\ \overline{1} \\
\rightarrow & (^{N^b \rightarrow N^b} HS\ ((\lambda x_1.(\lambda x_2.x_1))\ (SH^{N^a}\ \overline{0})))\ \overline{1} \\
\rightarrow & (\lambda x_4 : N.(^{N^b} HS\ (((\lambda x_1.(\lambda x_2.x_1))\ (SH^{N^a}\ \overline{0}))\ (SH^{N^b}\ x_4))))\ \overline{1} \\
\rightarrow & {}^{N^b} HS\ (((\lambda x_1.(\lambda x_2.x_1))\ (SH^{N^a}\ \overline{0}))\ (SH^{N^b}\ \overline{1})) \\
\rightarrow & {}^{N^b} HS\ ((\lambda x_2.(SH^{N^a}\ \overline{0}))\ (SH^{N^b}\ \overline{1})) \\
\rightarrow & {}^{N^b} HS\ (SH^{N^a}\ \overline{0}) \\
\rightarrow & \texttt{wrong}\ \text{“Parametricity violated”} \\
\rightarrow & \textbf{Error}\text{: “Parametricity violated”}
\end{array}
$$

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 insantiated 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.

$$
\begin{array}{ll}
 & (\lambda x_1.(\texttt{cons}\ (x_1\ \overline{0})\ (x_1\ \texttt{nil})))\ (SH^{\forall X.(X\to X)}\ \Lambda X.(\lambda x_2 : X.x_2)) \\
\to & (\lambda x_1.(\texttt{cons}\ (x_1\ \overline{0})\ (x_1\ \texttt{nil})))\ (SH^{L\to L}\ ((\Lambda X.(\lambda x_2 : X.x_2))\ \{L\})) \\
\to & (\lambda x_1.(\texttt{cons}\ (x_1\ \overline{0})\ (x_1\ \texttt{nil})))\ (SH^{L\to L}\ \lambda x_2 : L.x_2) \\
\to & (\lambda x_1.(\texttt{cons}\ (x_1\ \overline{0})\ (x_1\ \texttt{nil})))\ (\lambda x_3.(SH^{L}\ ((\lambda x_2 : L.x_2)\ ({}^{L}HS\ x_3)))) \\
\to & \texttt{cons}\ ((\lambda x_3.(SH^{L}\ ((\lambda x_2 : L.x_2)\ ({}^{L}HS\ x_3))))\ \overline{0}) \\
 & \quad ((\lambda x_3.(SH^{L}\ ((\lambda x_2 : L.x_2)\ ({}^{L}HS\ x_3))))\ \texttt{nil}) \\
\to & \texttt{cons}\ (SH^{L}\ ((\lambda x_2 : L.x_2)\ ({}^{L}HS\ \overline{0}))) \\
 & \quad ((\lambda x_3.(SH^{L}\ ((\lambda x_2 : L.x_2)\ ({}^{L}HS\ x_3))))\ \texttt{nil}) \\
\to & \texttt{cons}\ (SH^{L}\ ({}^{L}HS\ \overline{0}))\ ((\lambda x_3.(SH^{L}\ ((\lambda x_2 : L.x_2)\ ({}^{L}HS\ x_3))))\ \texttt{nil}) \\
\to & \texttt{cons}\ \overline{0}\ ((\lambda x_3.(SH^{L}\ ((\lambda x_2 : L.x_2)\ ({}^{L}HS\ x_3))))\ \texttt{nil}) \\
\to & \texttt{cons}\ \overline{0}\ (SH^{L}\ ((\lambda x_2 : L.x_2)\ ({}^{L}HS\ \texttt{nil}))) \\
\to & \texttt{cons}\ \overline{0}\ (SH^{L}\ ({}^{L}HS\ \texttt{nil})) \\
\to & \texttt{cons}\ \overline{0}\ \texttt{nil}
\end{array}
$$

Figure 2.5: Polymorphic function converted to Scheme function

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

Figure 2.6: Haskell grammar and evaluation contexts

$$\frac{}{\vdash_H \mathtt{L}} \quad \frac{}{\vdash_H \mathtt{N}} \quad \frac{}{\Gamma, u_H \vdash_H u_H}$$

$$\frac{\Gamma \vdash_H t_H}{\Gamma \vdash_H \{t_H\}} \quad \frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_H t'_H}{\Gamma \vdash_H t_H \to t'_H} \quad \frac{\Gamma, u_H \vdash_H t_H}{\Gamma \vdash_H \forall u_H.t_H}$$

$$\frac{\Gamma \vdash_H t_H \quad \Gamma, x_H : t_H \vdash_H e_H : t'_H}{\Gamma \vdash_H (\lambda x_H : t_H.e_H) : t_H \to t'_H} \quad \frac{\Gamma, u_H \vdash_H e_H : t_H}{\Gamma \vdash_H \Lambda u_H.e_H : \forall u_H.t_H} \quad \frac{}{\vdash_H \overline{n} : \mathtt{N}}$$

$$\frac{\Gamma \vdash_H t_H :}{\Gamma \vdash_H \mathtt{nil}\ t_H : \{t_H\}} \quad \frac{\Gamma \vdash_H e_H : t_H \quad \Gamma \vdash_H e'_H : \{t_H\}}{\Gamma \vdash_H \mathtt{cons}\ e_H\ e'_H : \{t_H\}} \quad \frac{}{\Gamma, x_H : t_H \vdash_H x_H : t_H}$$

$$\frac{\Gamma \vdash_H e_H : t_H \to t'_H \quad \Gamma \vdash_H e'_H : t_H}{\Gamma \vdash_H e_H\ e'_H : t'_H} \quad \frac{\Gamma \vdash_H e_H : t_H \to t_H}{\Gamma \vdash_H \mathtt{fix}\ e_H : t_H}$$

$$\frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_H e_H : \forall u_H.t'_H}{\Gamma \vdash_H e_H\langle t_H\rangle : t'_H[t_H/u_H]} \quad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathtt{hd}\ e_H : t_H} \quad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathtt{tl}\ e_H : \{t_H\}}$$

$$\frac{\Gamma \vdash_H e_H : \mathtt{N} \quad \Gamma \vdash_H e'_H : \mathtt{N}}{\Gamma \vdash_H o\ e_H\ e'_H : \mathtt{N}} \quad \frac{\Gamma \vdash_H e_H : \{t_H\}}{\Gamma \vdash_H \mathtt{null?}\ e_H : \mathtt{N}} \quad \frac{\Gamma \vdash_H \lfloor k_H \rfloor \quad \Gamma \vdash_S e_S : \mathtt{TST}}{\Gamma \vdash_H \mathtt{hs}\ k_H\ e_S : \lfloor k_H \rfloor}$$

$$\frac{\Gamma \vdash_H e_H : \mathtt{N} \quad \Gamma \vdash_H e'_H : t_H \quad \Gamma \vdash_H e''_H : t_H}{\Gamma \vdash_H \mathtt{if0}\ e_H\ e'_H\ e''_H : t_H} \quad \frac{\Gamma \vdash_H t_H}{\Gamma \vdash_H \mathtt{wrong}\ t_H\ \mathit{string} : t_H}$$

$$\frac{\Gamma \vdash_H t_H \quad \Gamma \vdash_M t_M \quad \Gamma \vdash_M e_M : t'_M \quad t_H \doteq t_M \quad t_M = t'_M}{\Gamma \vdash_H \mathtt{hm}\ t_H\ t_M\ e_M : t_H}$$

Figure 2.7: Haskell typing rules

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

Figure 2.8: Haskell operational semantics

$$\mathscr{E}[\texttt{hm}\ t_H\ \texttt{L}\ (\texttt{mh}\ \texttt{L}\ t'_H\ e_H)]_H \rightarrow \mathscr{E}[e_H]\ (t_H = t'_H \text{ and } t_H \neq \texttt{L})$$

$$\mathscr{E}[\texttt{hm}\ t_H\ \texttt{L}\ (\texttt{mh}\ \texttt{L}\ t'_H\ e_H)]_H \rightarrow \mathscr{E}[\texttt{wrong}\ t_H\ \text{“Type mismatch”}]\ (t_H \neq t'_H \text{ and } t_H \neq \texttt{L})$$

$$\mathscr{E}[\texttt{hm}\ t_H\ \texttt{L}\ (\texttt{ms}\ \texttt{L}\ w_S)]_H \rightarrow \mathscr{E}[\texttt{wrong}\ t_H\ \text{“Bad value”}]\ (t_H \neq \texttt{L})$$

$$\mathscr{E}[\texttt{hm}\ \texttt{N}\ \texttt{N}\ \overline{n}]_H \rightarrow \mathscr{E}[\overline{n}]$$

$$\mathscr{E}[\texttt{hm}\ \{t_H\}\ \{t_M\}\ (\texttt{nil}\ t'_M)]_H \rightarrow \mathscr{E}[\texttt{nil}\ t_H]$$

$$\mathscr{E}[\texttt{hm}\ \{t_H\}\ \{t_M\}\ (\texttt{cons}\ v_M\ v'_M)]_H \rightarrow \mathscr{E}[\texttt{cons}\ (\texttt{hm}\ t_H\ t_M\ v_M)\ (\texttt{hm}\ \{t_H\}\ \{t_M\}\ v'_M)]$$

$$\mathscr{E}[\texttt{hm}\ (t_H \rightarrow t'_H)\ (t_M \rightarrow t'_M)\ (\lambda x_M : t''_M.e_M)]_H \rightarrow$$
$$\qquad \mathscr{E}[\lambda x_H : t_H.\texttt{hm}\ t'_H\ t'_M\ ((\lambda x_M : t''_M.e_M)\ (\texttt{mh}\ t_M\ t_H\ x_H))]$$

$$\mathscr{E}[\texttt{hm}\ (\forall u_H.t_H)\ (\forall u_M.t_M)\ (\Lambda u'_M.e_M)]_H \rightarrow \mathscr{E}[\Lambda u_H.\texttt{hm}\ t_H\ t_M[\texttt{L}/u_M]\ e_M[\texttt{L}/u'_M]]$$

Figure 2.9: Haskell-ML operational semantics

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

Figure 2.10: Haskell-Scheme operational semantics

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

Figure 2.11: ML grammar and evaluation contexts

$$\frac{}{\vdash_M \mathtt{L}} \quad \frac{}{\vdash_M \mathtt{N}} \quad \frac{}{\Gamma, u_M \vdash_M u_M}$$

$$\frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \{t_M\}} \quad \frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_M t'_M}{\Gamma \vdash_M t_M \to t'_M} \quad \frac{\Gamma, u_M \vdash_M t_M}{\Gamma \vdash_M \forall u_M.t_M}$$

$$\frac{\Gamma \vdash_M t_M \quad \Gamma, x_M : t_M \vdash_M e_M : t'_M}{\Gamma \vdash_M (\lambda x_M : t_M.e_M) : t_M \to t'_M} \quad \frac{\Gamma, u_M \vdash_M e_M : t_M}{\Gamma \vdash_M \Lambda u_M.e_M : \forall u_M.t_M} \quad \frac{}{\vdash_M \overline{n} : \mathtt{N}}$$

$$\frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \mathtt{nil}\ t_M : \{t_M\}} \quad \frac{\Gamma \vdash_M e_M : t_M \quad \Gamma \vdash_M e'_M : \{t_M\}}{\Gamma \vdash_M \mathtt{cons}\ e_M\ e'_M : \{t_M\}} \quad \frac{}{\Gamma, x_M : t_M \vdash_M x_M : t_M}$$

$$\frac{\Gamma \vdash_M e_M : t_M \to t'_M \quad \Gamma \vdash_M e'_M : t_M}{\Gamma \vdash_H e_M\ e'_M : t'_M} \quad \frac{\Gamma \vdash_M e_M : t_M \to t_M}{\Gamma \vdash_M \mathtt{fix}\ e_M : t_M}$$

$$\frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_M e_M : \forall u_M.t'_M}{\Gamma \vdash_M e_M\langle t_M\rangle : t'_M[t_M/u_M]} \quad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathtt{hd}\ e_M : t_M} \quad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathtt{tl}\ e_M : \{t_M\}}$$

$$\frac{\Gamma \vdash_M e_M : \mathtt{N} \quad \Gamma \vdash_M e'_M : \mathtt{N}}{\Gamma \vdash_M o\ e_M\ e'_M : \mathtt{N}} \quad \frac{\Gamma \vdash_M e_M : \{t_M\}}{\Gamma \vdash_M \mathtt{null?}\ e_M : \mathtt{N}} \quad \frac{\Gamma \vdash_M \lfloor k_M \rfloor \quad \Gamma \vdash_S e_S : \mathtt{TST}}{\Gamma \vdash_M \mathtt{ms}\ k_M\ e_S : \lfloor k_M \rfloor}$$

$$\frac{\Gamma \vdash_M e_M : \mathtt{N} \quad \Gamma \vdash_M e'_M : t_M \quad \Gamma \vdash_M e''_M : t_M}{\Gamma \vdash_M \mathtt{if0}\ e_M\ e'_M\ e''_M : t_M} \quad \frac{\Gamma \vdash_M t_M}{\Gamma \vdash_M \mathtt{wrong}\ t_M\ \mathit{string} : t_M}$$

$$\frac{\Gamma \vdash_M t_M \quad \Gamma \vdash_H t_H \quad \Gamma \vdash_H e_H : t'_H \quad t_M \doteq t_H \quad t_H = t'_H}{\Gamma \vdash_M \mathtt{mh}\ t_M\ t_H\ e_H : t_M}$$

Figure 2.12: ML typing rules

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

Figure 2.13: ML operational semantics

$$\mathscr{E}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ t'_M\ w_M)]_M \to \mathscr{E}[w_M]\ (t_M = t'_M \text{ and } t_M \neq \texttt{L})$$
$$\mathscr{E}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hm}\ \texttt{L}\ t'_M\ w_M)]_M \to \mathscr{E}[\texttt{wrong}\ t_M\ \text{“Type mismatch”}]\ (t_M \neq t'_M \text{ and } t_M \neq \texttt{L})$$
$$\mathscr{E}[\texttt{mh}\ t_M\ \texttt{L}\ (\texttt{hs}\ \texttt{L}\ w_S)]_H \to \mathscr{E}[\texttt{wrong}\ t_M\ \text{“Bad value”}]\ (t_M \neq \texttt{L})$$
$$\mathscr{E}[\texttt{mh}\ \texttt{N}\ \texttt{N}\ \overline{n}]_M \to \mathscr{E}[\overline{n}]$$
$$\mathscr{E}[\texttt{mh}\ \{t_M\}\ \{t_H\}\ (\texttt{nil}\ t'_H)]_M \to \mathscr{E}[\texttt{nil}\ t_M]$$
$$\mathscr{E}[\texttt{mh}\ \{t_M\}\ \{t_H\}\ (\texttt{cons}\ e_H\ e'_H)]_M \to \mathscr{E}[\texttt{cons}\ (\texttt{mh}\ t_M\ t_H\ e_H)\ (\texttt{mh}\ \{t_M\}\ \{t_H\}\ e'_H)]$$
$$\mathscr{E}[\texttt{mh}\ (t_M \to t'_M)\ (t_H \to t'_H)\ (\lambda x_H : t''_H.e_H)]_M \to$$
$$\quad \mathscr{E}[\lambda x_M : t_M.\texttt{mh}\ t'_M\ t'_H\ ((\lambda x_H : t''_H.e_H)\ (\texttt{hm}\ t_H\ t_M\ x_M))]$$
$$\mathscr{E}[\texttt{mh}\ (\forall u_M.t_M)\ (\forall u_H.t_H)\ (\Lambda u'_H.e_H)]_M \to \mathscr{E}[\Lambda u_M.\texttt{mh}\ t_M\ t_H[\texttt{L}/u_H]\ e_H[\texttt{L}/u'_H]]$$

Figure 2.14: ML-Haskell operational semantics

$$\mathscr{E}[\texttt{ms N}\ \overline{n}]_M \rightarrow \mathscr{E}[\overline{n}]$$
$$\mathscr{E}[\texttt{ms N}\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong N}\ \text{“Not a number”}]\ (w_S \neq \overline{n})$$
$$\mathscr{E}[\texttt{ms}\ \{k_M\}\ \texttt{nil}]_M \rightarrow \mathscr{E}[\texttt{nil}\ \lfloor k_M \rfloor]$$
$$\mathscr{E}[\texttt{ms}\ \{k_M\}\ (\texttt{cons}\ v_S\ v'_S)]_M \rightarrow \mathscr{E}[\texttt{cons}\ (\texttt{ms}\ k_M\ v_S)\ (\texttt{ms}\ \{k_M\}\ v'_S)]$$
$$\mathscr{E}[\texttt{ms}\ \{k_M\}\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \lfloor \{k_M\} \rfloor\ \text{“Not a list”}]$$
$$(w_S \neq \texttt{nil}\ \text{and}\ w_S \neq \texttt{cons}\ v_S\ v'_S)$$
$$\mathscr{E}[\texttt{ms}\ (b \diamond t_M)\ (\texttt{sm}\ (b \diamond t_M)\ v_M)]_M \rightarrow \mathscr{E}[v_M]$$
$$\mathscr{E}[\texttt{ms}\ (b \diamond t_M)\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \lfloor b \diamond t_M \rfloor\ \text{“Brand mismatch”}]$$
$$(w_S \neq \texttt{sm}\ (b \diamond t_M)\ e_M)$$
$$\mathscr{E}[\texttt{ms}\ (k_M \rightarrow k'_M)\ (\lambda x_S.e_S)]_M \rightarrow$$
$$\mathscr{E}[\lambda x_M : \lfloor k_M \rfloor.\texttt{ms}\ k'_M\ ((\lambda x_S.e_S)\ (\texttt{sm}\ k_M\ x_M))]$$
$$\mathscr{E}[\texttt{ms}\ (k_M \rightarrow k'_M)\ w_S]_M \rightarrow \mathscr{E}[\texttt{wrong}\ \lfloor k_M \rightarrow k'_M \rfloor\ \text{“Not a function”}]$$
$$(w_S \neq \lambda x_S.e_S)$$
$$\mathscr{E}[\texttt{ms}\ (\forall u_M.k_M)\ w_S]_M \rightarrow \mathscr{E}[\Lambda u_M.\texttt{ms}\ k_M\ w_S]$$

Figure 2.15: ML-Scheme operational semantics

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

Figure 2.16: Scheme grammar and evaluation contexts

$$\frac{}{\vdash_S \texttt{TST}}$$

$$\frac{\Gamma, x_S : \texttt{TST} \vdash_S e_S : \texttt{TST}}{\Gamma \vdash_S \lambda x_S.e_S : \texttt{TST}} \qquad \frac{}{\vdash_S \overline{n} : \texttt{TST}} \qquad \frac{}{\vdash_S \texttt{nil} : \texttt{TST}}$$

$$\frac{\Gamma \vdash_S e_S : \texttt{TST} \quad \Gamma \vdash_S e'_S : \texttt{TST}}{\Gamma \vdash_S \texttt{cons}\ e_S\ e'_S : \texttt{TST}} \qquad \frac{}{\Gamma, x_S : \texttt{TST} \vdash_S x_S : \texttt{TST}}$$

$$\frac{\Gamma \vdash_S e_S : \texttt{TST} \quad \Gamma \vdash_S e'_S : \texttt{TST}}{\Gamma \vdash_H e_S\ e'_S : \texttt{TST}} \qquad \frac{\Gamma \vdash_S e_S : \texttt{TST}}{\Gamma \vdash_S f\ e_S : \texttt{TST}}$$

$$\frac{\Gamma \vdash_S e_S : \texttt{TST} \quad \Gamma \vdash_S e'_S : \texttt{TST}}{\Gamma \vdash_S o\ e_S\ e'_S : \texttt{TST}} \qquad \frac{\Gamma \vdash_S e_S : \texttt{TST}}{\Gamma \vdash_S p\ e_S : \texttt{TST}}$$

$$\frac{\Gamma \vdash_S e_S : \texttt{TST} \quad \Gamma \vdash_S e'_S : \texttt{TST} \quad \Gamma \vdash_S e''_S : \texttt{TST}}{\Gamma \vdash_S \texttt{if0}\ e_S\ e'_S\ e''_S : \texttt{TST}} \qquad \frac{}{\vdash_S \texttt{wrong}\ \mathit{string} : \texttt{TST}}$$

$$\frac{\Gamma \vdash_H \lfloor k_H \rfloor \quad \Gamma \vdash_H e_H : t_H \quad \lfloor k_H \rfloor = t_H}{\Gamma \vdash_S \texttt{sh}\ k_H\ e_H : \texttt{TST}} \qquad \frac{\Gamma \vdash_M \lfloor k_M \rfloor \quad \Gamma \vdash_M e_M : t_M \quad \lfloor k_M \rfloor = t_M}{\Gamma \vdash_S \texttt{sm}\ k_M\ e_M : \texttt{TST}}$$

Figure 2.17: Scheme typing rules

$$\mathscr{E}[(\lambda x_S.e_S)\ v_S]_S \rightarrow \mathscr{E}[e_S[v_S/x_S]]$$

$$\mathscr{E}[w_S\ v_S]_S \rightarrow \mathscr{E}[\texttt{wrong}\text{ ``Not a function''}]\ (w_S \neq \lambda x_S.e_S)$$

$$\mathscr{E}[+\ \overline{n}\ \overline{n}']_S \rightarrow \mathscr{E}[\overline{n+n'}]$$

$$\mathscr{E}[-\ \overline{n}\ \overline{n}']_S \rightarrow \mathscr{E}[\overline{max(n-n',0)}]$$

$$\mathscr{E}[o\ w_S\ w'_S]_S \rightarrow \mathscr{E}[\texttt{wrong}\text{ ``Not a number''}]\ (w_S \neq \overline{n}\text{ or }w'_S \neq \overline{n})$$

$$\mathscr{E}[\texttt{if0}\ \overline{0}\ e_S\ e'_S]_S \rightarrow \mathscr{E}[e_S]$$

$$\mathscr{E}[\texttt{if0}\ \overline{n}\ e_S\ e'_S]_S \rightarrow \mathscr{E}[e'_S]\ (n \neq 0)$$

$$\mathscr{E}[\texttt{if0}\ w_S\ e_S\ e'_S]_S \rightarrow \mathscr{E}[\texttt{wrong}\text{ ``Not a number''}]\ (w_S \neq \overline{n})$$

$$\mathscr{E}[f\ \texttt{nil}]_S \rightarrow \mathscr{E}[\texttt{wrong}\text{ ``Empty list''}]$$

$$\mathscr{E}[\texttt{hd}\ (\texttt{cons}\ v_S\ v'_S)]_S \rightarrow \mathscr{E}[v_S]$$

$$\mathscr{E}[\texttt{tl}\ (\texttt{cons}\ v_S\ v'_S)]_S \rightarrow \mathscr{E}[v'_S]$$

$$\mathscr{E}[f\ w_S]_S \rightarrow \mathscr{E}[\texttt{wrong}\text{ ``Not a list''}]\ (w_S \neq \texttt{nil}\text{ and }w_S \neq \texttt{cons}\ v_S\ v'_S)$$

$$\mathscr{E}[\texttt{fun?}\ (\lambda x_S.e_S)]_S \rightarrow \mathscr{E}[\overline{0}]$$

$$\mathscr{E}[\texttt{fun?}\ w_S]_S \rightarrow \mathscr{E}[\overline{1}]\ (w_S \neq \lambda x_S.e_S)$$

$$\mathscr{E}[\texttt{list?}\ \texttt{nil}]_S \rightarrow \mathscr{E}[\overline{0}]$$

$$\mathscr{E}[\texttt{list?}\ (\texttt{cons}\ v_S\ v'_S)]_S \rightarrow \mathscr{E}[\overline{0}]$$

$$\mathscr{E}[\texttt{list?}\ w_S]_S \rightarrow \mathscr{E}[\overline{1}]\ (w_S \neq \texttt{nil}\text{ and }w_S \neq \texttt{cons}\ v_S\ v'_S)$$

$$\mathscr{E}[\texttt{null?}\ \texttt{nil}]_S \rightarrow \mathscr{E}[\overline{0}]$$

$$\mathscr{E}[\texttt{null?}\ w_S]_S \rightarrow \mathscr{E}[\overline{1}]\ (w_S \neq \texttt{nil})$$

$$\mathscr{E}[\texttt{num?}\ \overline{n}]_S \rightarrow \mathscr{E}[\overline{0}]$$

$$\mathscr{E}[\texttt{num?}\ w_S]_S \rightarrow \mathscr{E}[\overline{1}]\ (w_S \neq \overline{n})$$

$$\mathscr{E}[\texttt{wrong}\ \textit{string}]_S \rightarrow \textbf{Error: }\textit{string}$$

Figure 2.18: Scheme operational semantics

$$\mathscr{E}[\texttt{sh L (hm L } k_M\ w_M)]_S \rightarrow \mathscr{E}[\texttt{wrong} \text{ ``Bad value''}]$$
$$\mathscr{E}[\texttt{sh L (hs L } w_S)]_S \rightarrow \mathscr{E}[w_S]$$
$$\mathscr{E}[\texttt{sh N } \overline{n}]_S \rightarrow \mathscr{E}[\overline{n}]$$
$$\mathscr{E}[\texttt{sh } \{k_H\}\ (\texttt{nil } t_H)]_S \rightarrow \mathscr{E}[\texttt{nil}]$$
$$\mathscr{E}[\texttt{sh } \{k_H\}\ (\texttt{cons } e_H\ e'_H)]_S \rightarrow \mathscr{E}[\texttt{cons } (\texttt{sh } k_H\ e_H)\ (\texttt{sh } \{k_H\}\ e'_H)]$$
$$\mathscr{E}[\texttt{sh } (k_H \rightarrow k'_H)\ (\lambda x_H : t_H.e_H)]_S \rightarrow$$
$$\qquad \mathscr{E}[\lambda x_S.\texttt{sh } k'_H\ ((\lambda x_H : t_H.e_H)\ (\texttt{hs } k_H\ x_S))]$$
$$\mathscr{E}[\texttt{sh } (\forall u_H.k_H)\ (\Lambda u'_H.e_H)]_S \rightarrow \mathscr{E}[\texttt{sh } k_H[\texttt{L}/u_H]\ e_H[\texttt{L}/u'_H]]$$

Figure 2.19: Scheme-Haskell operational semantics

$$\mathscr{E}[\texttt{sm}\ \texttt{L}\ (\texttt{mh}\ \texttt{L}\ k_H\ e_H)]_S \rightarrow \mathscr{E}[\texttt{wrong}\ \text{“Bad value”}]$$

$$\mathscr{E}[\texttt{sm}\ \texttt{L}\ (\texttt{ms}\ \texttt{L}\ w_S)]_S \rightarrow \mathscr{E}[w_S]$$

$$\mathscr{E}[\texttt{sm}\ \texttt{N}\ \overline{n}]_S \rightarrow \mathscr{E}[\overline{n}]$$

$$\mathscr{E}[\texttt{sm}\ \{k_M\}\ (\texttt{nil}\ t_M)]_S \rightarrow \mathscr{E}[\texttt{nil}]$$

$$\mathscr{E}[\texttt{sm}\ \{k_M\}\ (\texttt{cons}\ v_M\ v'_M)]_S \rightarrow \mathscr{E}[\texttt{cons}\ (\texttt{sm}\ k_M\ v_M)\ (\texttt{sm}\ \{k_M\}\ v'_M)]$$

$$\mathscr{E}[\texttt{sm}\ (k_M \rightarrow k'_M)\ (\lambda x_M : t_M.e_M)]_S \rightarrow$$
$$\mathscr{E}[\lambda x_S.\texttt{sm}\ k'_M\ ((\lambda x_M : t_M.e_M)\ (\texttt{ms}\ k_M\ x_S))]$$

$$\mathscr{E}[\texttt{sm}\ (\forall u_M.k_M)\ (\Lambda u'_M.e_M)]_S \rightarrow \mathscr{E}[\texttt{sm}\ k_M[\texttt{L}/u_M]\ e_M[\texttt{L}/u'_M]]$$

Figure 2.20: Scheme-ML operational semantics

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

Figure 2.21: Unbrand function

$$
\begin{gathered}
x \doteq x \\
x \doteq y \Rightarrow y \doteq x \\
x \doteq y \text{ and } y \doteq z \Rightarrow x \doteq z \\
t_H \doteq \mathtt{L} \\
t_M \doteq \mathtt{L} \\
t_H = t_M \Rightarrow t_H \doteq t_M
\end{gathered}
$$

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 [7], which was based on proofs by Pierce [11] and Matthews and Findler [8]. 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 error. 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. Straightforward 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* $\vdash_H \overline{n} : t_H$ *then* $t_H = \mathtt{N}$.
4. *If* $\Gamma \vdash_H \mathtt{nil}\ t_H : t'_H$ *then* $t'_H = \{t_H\}$ *and* $\Gamma \vdash_H t_H$.
5. *If* $\Gamma \vdash_H \mathtt{cons}\ e_H\ e'_H : t_H$ *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* $\Gamma \vdash_H \mathtt{fix}\ e_H : t_H$ *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* $\Gamma \vdash_H \texttt{hd}\, e_H : t_H$ *then* $\Gamma \vdash_H e_H : \{t_H\}$.

11. *If* $\Gamma \vdash_H \texttt{tl}\, e_H : t_H$ *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* $t_H = \texttt{N}$, $\Gamma \vdash_H e_H : \texttt{N}$, *and* $\Gamma \vdash_H e'_H : \texttt{N}$.

13. *If* $\Gamma \vdash_H \texttt{null?}\, e_H : t_H$ *then* $t_H = \texttt{N}$ *and* $\Gamma \vdash_H e_H : \{t'_H\}$.

14. *If* $\Gamma \vdash_H \texttt{if0}\, e_H\; e'_H\; e''_H : t_H$ *then* $\Gamma \vdash_H e_H : \texttt{N}$, $\Gamma \vdash_H e'_H : t_H$, $\Gamma \vdash_H e''_H : t_H$.

15. *If* $\Gamma \vdash_H \texttt{wrong}\, t_H$ *string* $: t'_H$ *then* $t'_H = t_H$.

16. *If* $\Gamma \vdash_H \texttt{hm}\, t_H\; t_M\; e_M : t'_H$ *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* $\Gamma \vdash_H \texttt{hs}\, k_H\; e_S : t_H$ *then* $t_H = \lfloor k_H \rfloor$, $\Gamma \vdash_H \lfloor k_H \rfloor$, *and* $\Gamma \vdash_S e_S : \texttt{TST}$.

18. *If* $\Gamma \vdash_S \texttt{sh}\, k_H\; e_H : \texttt{TST}$ *then* $\Gamma \vdash_H \lfloor k_H \rfloor$, $\Gamma \vdash_H e_H : t_H$, *and* $\lfloor k_H \rfloor = t_H$.

19. *If* $\Gamma \vdash_S \texttt{sm}\, k_M\; e_M : \texttt{TST}$ *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* $\Gamma \vdash_H v_H : \mathtt{L}$ *then* $v_H \in \{\mathtt{hm}\,\mathtt{L}\,t_M\ w_M, \mathtt{hs}\,\mathtt{L}\,w_S\}$.
2. *If* $\Gamma \vdash_H v_H : \mathtt{N}$ *then* $v_H = \overline{n}$.
3. *If* $\Gamma \vdash_H v_H : \{t_H\}$ *then* $v_H \in \{\mathtt{nil}\,t_H, \mathtt{cons}\ e_H\ e'_H\}$.
4. *If* $\Gamma \vdash_H v_H : t_H \to t'_H$ *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* $\Gamma \vdash_M w_M : \mathtt{L}$ *then* $w_M \in \{\mathtt{mh}\,\mathtt{L}\,t_M\ w_M, \mathtt{ms}\,\mathtt{L}\,w_S\}$.
7. *If* $\Gamma \vdash_M w_M : \mathtt{N}$ *then* $w_M = \overline{n}$.
8. *If* $\Gamma \vdash_M w_M : \{t_M\}$ *then* $w_M \in \{\mathtt{nil}\,t_M, \mathtt{cons}\ v_M\ v'_M\}$.
9. *If* $\Gamma \vdash_M w_M : t_M \to t'_M$ *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* $e_H \to e'_H$ *or* $e_H \to$ **Error:** *string*.

Proof. By structural induction on e_H.

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. $\overline{n}$

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

Case 1.5. `nil` t_H

`nil` t_H *is an unforced value.*

Case 1.6. `cons` $e_H\ e'_H$

`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* $e_H \rightarrow$ **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* $e_H \rightarrow$ **Error:** *string then* $e_H\ e'_H \rightarrow$ **Error:** *string.*

Case 1.8. $e_H\langle t_H\rangle$

e_H *is an unforced value or* $e_H \rightarrow e'_H$ *or* $e_H \rightarrow$ **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* $e_H \rightarrow$ **Error:** *string then* $e_H\langle t_H\rangle \rightarrow$ **Error:** *string.*

Case 1.9. $\texttt{fix}\, e_H$

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

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

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

Case 1.11. $\texttt{if0}\, e_H\, e'_H\, e''_H$

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

Case 1.12. $f\ e_H$

e_H *is an unforced value or* $e_H \to e'_H$ *or* $e_H \to$ **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*

$e_H \in \{\texttt{nil}\ t_H, \texttt{cons}\ e'_H\ e''_H\}$ *by lemma 3.* $\texttt{hd}\ (\texttt{nil}\ t_H) \to \texttt{wrong}\ t_H$ "Empty list". $\texttt{tl}\ (\texttt{nil}\ t_H) \to \texttt{wrong}\ \{t_H\}$ "Empty list". $\texttt{hd}\ (\texttt{cons}\ e'_H\ e''_H) \to e'_H$. $\texttt{tl}\ (\texttt{cons}\ e'_H\ e''_H) \to e''_H$. *If* $e_H \to e'_H$ *then* $f\ e_H \to f\ e'_H$. *If* $e_H \to$ **Error:** *string then* $f\ e_H \to$ **Error:** *string.*

Case 1.13. $\texttt{null?}\ e_H$

e_H is an unforced value or $e_H \to e'_H$ or $e_H \to$ **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 $e_H \in \{\texttt{nil}\ t_H, \texttt{cons}\ e'_H\ e''_H\}$ by lemma 3.* $\texttt{null?}\ (\texttt{nil}\ t_H) \to \overline{0}$. $\texttt{null?}\ (\texttt{cons}\ e'_H\ e''_H) \to \overline{1}$. *If $e_H \to e'_H$ then* $\texttt{null?}\ e_H \to \texttt{null?}\ e'_H$. *If $e_H \to$* **Error:** *string then* $\texttt{null?}\ e_H \to$ **Error:** *string.*

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

$\texttt{wrong}\ t_H$ *string* $\to$ **Error:** *string.*

Case 1.15. $\texttt{hm}\ t_H\ t_M\ e_M$

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

Case 1.15.1. $t_H = \texttt{L}$

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

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

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

Case 1.15.3. $t_H = \texttt{N}$ *and* $t_M = \texttt{N}$

$\vdash_M e_M : \mathtt{N}$ *by lemmas 1 and 2 and* $e_M = \overline{n}$ *by lemma 3.* $\mathtt{hm}\,\mathtt{N}\,\mathtt{N}\,\overline{n} \to \overline{n}$.

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* $e_M \in \{\mathtt{nil}\ t''_M, \mathtt{cons}\ v_M\ v'_M\}$ *by lemma 3.* $\mathtt{hm}\ \{t'_H\}\ \{t'_M\}\ (\mathtt{nil}\ t''_M) \to \mathtt{nil}\ t'_H$. $\mathtt{hm}\ \{t'_H\}\ \{t'_M\}\ (\mathtt{cons}\ v_M\ v'_M) \to \mathtt{cons}\ (\mathtt{hm}\ t'_H\ t'_M\ v_M)\ (\mathtt{hm}\ \{t'_H\}\ \{t'_M\}\ v'_M)$.

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

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

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.* $\mathtt{hm}\ (\forall u_H.t'_H)\ (\forall u_M.t'_M)\ (\Lambda u'_M.e'_M) \to \Lambda u_H.\mathtt{hm}\ t'_H\ t'_M[\mathtt{L}/u_M]\ e'_M[\mathtt{L}/u'_M]$.

Case 1.16. $\mathtt{hs}\ k_H\ e_S$

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

Case 1.16.1. $\mathtt{L}$

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

Case 1.16.2. $\mathtt{N}$

$\mathtt{hs}\ \mathtt{N}\ \overline{n} \to \overline{n}$. $\mathtt{hs}\ \mathtt{N}\ e_S \to \mathtt{wrong}\ \mathtt{N}$ "Not a number" $(e_S \neq \overline{n})$.

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

$\texttt{hs}\ \{k'_H\}\ \texttt{nil} \rightarrow \texttt{nil}\ \lfloor k'_H \rfloor$. $\texttt{hs}\ \{k'_H\}\ (\texttt{cons}\ v_S\ v'_S) \rightarrow \texttt{cons}\ (\texttt{hs}\ k_H\ v_S)\ (\texttt{hs}\ \{k'_H\}\ v'_S)$. $\texttt{hs}\ \{k'_H\}\ e_S \rightarrow \texttt{wrong}\ \lfloor\{k'_H\}\rfloor$ "Not a list" $(e_S \neq \texttt{nil}$ *and* $e_S \neq \texttt{cons}\ v_S\ v'_S)$.

Case 1.16.4. $b \diamond t_H$

$\texttt{hs}\ (b\diamond t_H)\ (\texttt{sh}\ (b\diamond t_H)\ e_H) \rightarrow e_H$. $\texttt{hs}\ (b\diamond t_H)\ e_S \rightarrow \texttt{wrong}\ t_H$ "Brand mismatch" $(e_S \neq \texttt{sh}\ (b \diamond t_H)\ e_H)$.

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

$\texttt{hs}\ (k'_H \rightarrow k''_H)\ (\lambda x_S.e_S) \rightarrow \lambda x_H : \lfloor k'_H \rfloor.\texttt{hs}\ k''_H\ ((\lambda x_S.e_S)\ (\texttt{sh}\ k'_H\ x_H))$. $\texttt{hs}\ (k'_H \rightarrow k''_H)\ e_S \rightarrow \texttt{wrong}\ \lfloor k'_H \rightarrow k''_H \rfloor$ "Not a function" $(e_S \neq \lambda x_S.e_S)$.

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

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

$\square$

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* $e_M \rightarrow$ **Error:** *string.*

Proof. By structural induction on e_M. Cases similar to Haskell cases are omitted.

Case 2.1. $\texttt{mh}\ t_M\ t_H\ e_H$

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

TODO

Case 2.1.1.

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

Case 2.2. $e_M\ e'_M$

*e_M is an unforced value or $e_M \rightarrow e''_M$ or $e_M \rightarrow$ **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 $\rightarrow$ **Error:** string then $e_M\ e'_M \rightarrow$ **Error:** string. e'_M is an unforced value or e'_M $\rightarrow e''_M$ or $e'_M \rightarrow$ **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 $e'_M \rightarrow$ **Error:** string and e_M is an unforced value then $e_M\ e'_M \rightarrow$ **Error:** string. If e_M and e'_M are unforced values then $e_M \rightarrow e'_M$.*

Case 2.3. `cons` $e_M\ e'_M$

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

□

Theorem 3. *Scheme Progress*

*If $\vdash_S e_S :$ `TST` then e_S is an unforced value or $e_S \rightarrow e'_S$ or $e_S \rightarrow$ **Error:** string.*

Proof. By structural induction on e_S. 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. $\texttt{sm}\ k_M\ e_M$

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

Case 3.2.1. $\texttt{L}$

$e_M \in \{\texttt{mh}\ \texttt{L}\ k_H\ e_H, \texttt{ms}\ \texttt{L}\ w_S\}$ *by lemma 3.* $\texttt{sm}\ \texttt{L}\ (\texttt{mh}\ \texttt{L}\ k_H\ e_H) \to \texttt{wrong}$ "Bad value". $\texttt{sm}\ \texttt{L}\ (\texttt{ms}\ \texttt{L}\ w_S) \to w_S$.

Case 3.2.2. $\texttt{N}$

$e_M = \overline{n}$ *by lemma 3.* $\texttt{sm}\ \texttt{N}\ \overline{n} \to \overline{n}$.

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

$e_M \in \{\texttt{nil}\ t_M, \texttt{cons}\ v_M\ v'_M\}$ *by lemma 3.* $\texttt{sm}\ \{k'_M\}\ (\texttt{nil}\ t_M) \to \texttt{nil}$. $\texttt{sm}\ \{k'_M\}\ (\texttt{cons}\ v_M\ v'_M) \to \texttt{cons}\ (\texttt{sm}\ k'_M\ v_M)\ (\texttt{sm}\ \{k'_M\}\ v'_M)$.

Case 3.2.4. $k'_M \to k''_M$

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

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

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

Case 3.2.6. $b \diamond t_M$

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

$\square$

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. Inversion (Lemma **??**) and uniqueness of types (Lemma **??**) are used to 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_A^1 is substituted for free occurrences of x within e_A^2, e_A^1 and x have the same type, and the result has the same type as e_A^2, where $A \in \{H, M, S\}$.

Lemma 4. *Expression Substitution Preservation*

If $\Gamma, x_1 : T_1 \vdash_A e_A : T_2$ *and* $\Gamma \vdash_A x_2 : T_1$ *then* $\Gamma \vdash_A e_A[x_2/x_1] : T_2$ *where* $A \in \{H, M\}$*. If* $\Gamma, x_1 : TST \vdash_S e_S : TST$ *and* $\Gamma \vdash_S x_2 : TST$ *then* $\Gamma \vdash_S e_S[x_2/x_1] : TST$*.*

Proof. By structural induction.

□

If T_1 is substituted for free occurrences of X within e_A of type T_2, the type of the result is T_1 substituted for free occurrences of X within T_2, where $A \in \{H, M\}$.

Lemma 5. *Type Substitution Preservation*

If $\Gamma, X \vdash_A e_A : T_1$ *and* $\Gamma \vdash_A T_2$ *then* $\Gamma \vdash_A e_A[T_2/X] : T_1[T_2/X]$ *where* $A \in \{H, M\}$*.*

Proof. By structural induction.

□

Lemma 6. *Evaluation Context Preservation*

If $\Gamma \vdash_A e_A^1 : T_1$*,* $\Gamma \vdash_A e_A^2 : T_1$*, and* $\mathscr{E}[e_A^1] : T_2$ *then* $\mathscr{E}[e_A^2] : T_2$ *where* $A \in \{H, M, S\}$*.*

Proof. By structural induction.

$\square$

Theorem 4. *Preservation*

If $\Gamma \vdash_A e_A^1 : T$ *and* $e_A^1 \rightarrow e_A^2$ *then* $\Gamma \vdash_A e_A^2 : T$ *where* $A \in \{H, M\}$*. If* $\Gamma \vdash_S e_S^1 : TST$ *and* $e_S^1 \rightarrow e_S^2$ *then* $\Gamma \vdash_S e_S^2 : TST$*.*

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 4.1. $(\lambda x : T_1.e_A^1)\ e_A^2 \rightarrow e_A^1[e_A^2/x]$ *where* $A \in \{H, M\}$

$\Gamma \vdash_A (\lambda x : T_1.e_A^1)\ e_A^2 : T$ *by premise and uniqueness of types (Lemma **??**).* $\Gamma \vdash_A \lambda x : T_1.e_A^1 : T_1 \rightarrow T_2$*,* $\Gamma, x : T_1 \vdash_A e_A^1 : T_2$*,* $\Gamma \vdash_A e_A^2 : T_1$*,* $\Gamma, x : T_1 \vdash_A x : T_1$*, and* $T = T_2$ *by inversion (Lemma **??**) and uniqueness of types.* $\Gamma \vdash_A e_A^1[e_A^2/x] : T_2$ *by term substitution (Lemma **??**).* $\Gamma \vdash_A e_A^1[e_A^2/x] : T$ *because* $T_2 = T$*.*

Case 4.2. $\texttt{fix}\ (\lambda x : T_1.e_A) \rightarrow e_A[(\texttt{fix}\ (\lambda x : T_1.e_A))/x]$ *where* $A \in \{H, M\}$

$\Gamma \vdash_A \texttt{fix}\ (\lambda x : T_1.e_A) : T$ *by premise and uniqueness of types (Lemma **??**).* $\Gamma \vdash_A \lambda x : T_1.e_A : T_1 \rightarrow T_2$*,* $\Gamma, x : T_1 \vdash_A e_A : T_1$*,* $\Gamma, x : T_1 \vdash_A x : T_1$*, and* $T = T_1$ *by inversion (Lemma **??**) and uniqueness of types.* $\Gamma \vdash_A e_A[(\texttt{fix}\ (\lambda x : T_1.e_A))/x] : T_1$ *by term substitution (Lemma **??**).* $\Gamma \vdash_A e_A[(\texttt{fix}\ (\lambda x : T_1.e_A))/x] : T$ *because* $T_1 = T$*.*

Case 4.3. $(\Lambda X.e_A)\ \{T_1\} \rightarrow e_A[T_1/X]$ *where* $A \in \{H, M\}$

$\Gamma \vdash_A (\Lambda X.e_H)\ \{T_1\} : T$ *by premise and uniqueness of types (Lemma **??**).* $\Gamma \vdash_A \Lambda X.e_A : \forall X.T_2$*,* $\Gamma, X \vdash_A e_A : T_2$*, and* $T = T_2[T_1/X]$ *by inversion (Lemma **??**) and uniqueness of types.* $\Gamma \vdash_A e_A[T_1/X] : T_2[T_1/X]$ *by type substitution (Lemma **??**).* $\Gamma \vdash_A e_A[T_1/X] : T$ *because* $T_2[T_1/X] = T$*.*

Case 4.4. $\texttt{hd nil}^{T_1} \rightarrow \texttt{wrong}^{T_1}$ "Empty list" *where* $A \in \{H, M\}$

$\Gamma \vdash_A \texttt{hd nil}^{T_1} : T$ *by premise and uniqueness of types (Lemma **??**).* $\Gamma \vdash_A \texttt{nil}^{T_1} : [T_1]$*,* $T = T_1$*, and* $\Gamma \vdash_A \texttt{wrong}^{T_1}$ "Empty list" $: T_1$ *by inversion (Lemma **??**) and uniqueness of types.* $\Gamma \vdash_A \texttt{wrong}^{T_1}$ "Empty list" $: T$ *because* $T_1 = T$*.*

Case 4.5. $\texttt{tl nil}^{T_1} \rightarrow \texttt{wrong}^{[T_1]}$ "Empty list" *where* $A \in \{H, M\}$

$\Gamma \vdash_A \texttt{tl nil}^{T_1} : T$ *by premise and uniqueness of types (Lemma **??**).* $\Gamma \vdash_A \texttt{nil}^{T_1} : [T_1]$*,* $T = [T_1]$*, and* $\Gamma \vdash_A \texttt{wrong}^{[T_1]}$ "Empty list" $: [T_1]$ *by inversion (Lemma **??**) and uniqueness of types.* $\Gamma \vdash_A \texttt{wrong}^{[T_1]}$ "Empty list" $: T$ *because* $[T_1] = T$*.*

Case 4.6. $\texttt{hd}\ (\texttt{cons}\ e_H^1\ e_H^2) \rightarrow e_H^1$

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

Case 4.7. $\texttt{hd}\ (\texttt{cons}\ v_M^1\ v_M^2) \rightarrow v_M^1$

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

Case 4.8. $\texttt{tl}\ (\texttt{cons}\ e_H^1\ e_H^2) \rightarrow e_H^2$

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

and uniqueness of types. $\Gamma \vdash_H e_H^2 : T$ *because* $[T_1] = T$.

Case 4.9. $\texttt{tl}\ (\texttt{cons}\ v_M^1\ v_M^2) \to v_M^2$

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

Case 4.10. $\texttt{hd}\ ({}^{[T_1]}MH^{[T_1]}\ (\texttt{cons}\ e_H^1\ e_H^2)) \to {}^{T_1}MH^{T_1}\ e_H^1$

$\Gamma \vdash_M \texttt{hd}\ ({}^{[T_1]}MH^{[T_1]}\ (\texttt{cons}\ e_H^1\ e_H^2)) : T$ *by premise and uniqueness of types (Lemma* **??***).* $\Gamma \vdash_H e_H^1 : T_1$, $\Gamma \vdash_H \texttt{cons}\ e_H^1\ e_H^2 : [T_1]$, $\Gamma \vdash_M {}^{[T_1]}MH^{[T_1]}\ (\texttt{cons}\ e_H^1\ e_H^2) : [T_1]$, $T = T_1$, *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 4.11. $\texttt{hd}\ (SH^{[T_1]}\ (\texttt{cons}\ e_H^1\ e_H^2)) \to SH^{T_1}\ e_H^1$

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

Case 4.12. $\texttt{tl}\ ({}^{[T_1]}MH^{[T_1]}\ (\texttt{cons}\ e_H^1\ e_H^2)) \to {}^{[T_1]}MH^{[T_1]}\ e_H^2$

$\Gamma \vdash_M \texttt{tl}\ ({}^{[T_1]}MH^{[T_1]}\ (\texttt{cons}\ e_H^1\ e_H^2)) : T$ *by premise and uniqueness of types (Lemma* **??***).* $\Gamma \vdash_H e_H^2 : [T_1]$, $\Gamma \vdash_H \texttt{cons}\ e_H^1\ e_H^2 : [T_1]$, $\Gamma \vdash_M {}^{[T_1]}MH^{[T_1]}\ (\texttt{cons}\ e_H^1\ e_H^2) : [T_1]$, $T = [T_1]$, *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 4.13. $\texttt{tl}\ (SH^{[T_1]}\ (\texttt{cons}\ e_H^1\ e_H^2)) \to SH^{[T_1]}\ e_H^2$

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

Case 4.14. $+\ \overline{n_1}\ \overline{n_2} \to \overline{n_1 + n_2}$ *where* $A \in \{H, M\}$

$\vdash_A + \overline{n_1}\ \overline{n_2} : T$ *by premise and uniqueness of types (Lemma ??).* $\vdash_A \overline{n_1} : N$, $\vdash_A \overline{n_2} : N$, $T = N$, *and* $\vdash_A \overline{n_1 + n_2} : N$ *by inversion (Lemma ??) and uniqueness of types.* $\vdash_A \overline{n_1 + n_2} : T$ *because* $N = T$.

Case 4.15. $- \ \overline{n_1}\ \overline{n_2} \rightarrow \overline{max(n_1 - n_2, 0)}$ *where* $A \in \{H, M\}$

$\vdash_A - \ \overline{n_1}\ \overline{n_2} : T$ *by premise and uniqueness of types (Lemma ??).* $\vdash_A \overline{n_1} : N$, $\vdash_A \overline{n_2} : N$, $T = N$, *and* $\vdash_A \overline{max(n_1 - n_2, 0)} : N$ *by inversion (Lemma ??) and uniqueness of types.* $\vdash_A \overline{max(n_1 - n_2, 0)} : T$ *because* $N = T$.

Case 4.16. $\texttt{null?}\ \texttt{nil}^{T_1} \rightarrow \overline{0}$ *where* $A \in \{H, M\}$

$\vdash_A \texttt{null?}\ \texttt{nil}^{T_1} : T$ *by premise and uniqueness of types (Lemma ??).* $T = N$ *and* $\vdash_A \overline{0} : N$ *by inversion (Lemma ??) and uniqueness of types.* $\vdash_A \overline{0} : T$ *because* $N = T$.

Case 4.17. $\texttt{null?}\ (\texttt{cons}\ B_A^1\ B_A^2) \rightarrow \overline{1}$ *where* $(A, B) \in \{(H, e), (M, v)\}$

$\Gamma \vdash_A \texttt{null?}\ (\texttt{cons}\ B_A^1\ B_A^2) : T$ *by premise and uniqueness of types (Lemma ??).* $T = N$ *and* $\vdash_A \overline{1} : N$ *by inversion (Lemma ??) and uniqueness of types.* $\vdash_A \overline{1} : T$ *because* $N = T$.

Case 4.18. $\texttt{null?}\ ({}^{[T]}MH^{[T]}\ (\texttt{cons}\ e_H^1\ e_H^2)) \rightarrow \overline{1}$

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

Case 4.19. $\texttt{if0}\ \overline{0}\ e_A^1\ e_A^2 \rightarrow e_A^1$ *where* $A \in \{H, M\}$

$\Gamma \vdash_A \texttt{if0}\ \overline{0}\ e_A^1\ e_A^2 : T$ *by premise and uniqueness of types (Lemma ??).* $\Gamma \vdash_A e_A^1 : T_1$ *and* $T = T_1$ *by inversion (Lemma ??) and uniqueness of types.* $\Gamma \vdash_A e_A^1 : T$ *because* $T_1 = T$.

Case 4.20. $\texttt{if0}\ \overline{n}\ e_A^1\ e_A^2 \rightarrow e_A^2\ (n \neq 0)$ *where* $A \in \{H, M\}$

$\Gamma \vdash_A$ `if0` $\overline{n}\ e_A^1\ e_A^2 : T$ *by premise and uniqueness of types (Lemma ??).* $\Gamma \vdash_A e_A^2 : T_1$ *and* $T = T_1$ *by inversion (Lemma ??) and uniqueness of types.* $\Gamma \vdash_A e_A^2 : T$ *because* $T_1 = T$.

Case 4.21. $\texttt{wrong}^{T_1}$ string $\rightarrow$ **Error**: string

Irrelevant because an error terminates the computation.

Case 4.22. `force` $v_S \rightarrow v_S$ ($v_M \neq {}^{T}MH^{T}\ e_H$ *or* $v_M = {}^{[T]}MH^{[T]}$ (`cons` $e_H^1\ e_H^2$))

$\vdash_M$ `force` $v_M : T$ *by premise and uniqueness of types (Lemma ??).* $\vdash_M v_M : T$ *by inversion (Lemma ??) and uniqueness of types.*

Case 4.23. ${}^{L}AB^{L}\ ({}^{L}BS\ v_S) \rightarrow {}^{L}AS\ v_S$ *where* $(A, B) \in \{(H, M), (M, H)\}$

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

Case 4.24. ${}^{N}AB^{N}\ \overline{n} \rightarrow \overline{n}$ *where* $(A, B) \in \{(H, M), (M, H)\}$

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

Case 4.25. ${}^{N}AS\ \overline{n} \rightarrow \overline{n}$ *where* $A \in \{H, M\}$

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

Case 4.26. ${}^{N}AS\ v_S \rightarrow {}^{N}AS$ (`wrong` "Not a number") ($v_S \neq \overline{n}$) *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.* $\vdash_S$ `wrong` "Not a number" $: TST$

by inversion. $\vdash_A {}^{N}AS$ (`wrong` "Not a number") $: N$ *by inversion and uniqueness of types.* $\vdash_A {}^{N}AS$ (`wrong` "Not a number") $: T$ *because* $N = T$.

Case 4.27. ${}^{[T_1]}AB^{[T_1]}\ \texttt{nil}^{T_1} \to \texttt{nil}^{T_1}$ *where* $(A, B) \in \{(H, M), (M, H)\}$

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

Case 4.28. ${}^{[T_1]}AS\ \texttt{nil} \to \texttt{nil}^{T_1}$ *where* $A \in \{H, M\}$

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

Case 4.29. ${}^{[T_1]}HM^{[T_1]}\ (\texttt{cons}\ v_M^1\ v_M^2) \to \texttt{cons}\ ({}^{T_1}HM^{T_1}\ v_M^1)\ ({}^{[T_1]}HM^{[T_1]}\ v_M^2)$

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

Case 4.30. ${}^{[T_1]}AS\ (\texttt{cons}\ v_S^1\ v_S^2) \to \texttt{cons}\ ({}^{T_1}AS\ v_S^1)\ ({}^{[T_1]}AS\ v_S^2)$ *where* $A \in \{H, M\}$

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

Case 4.31. $SM^{[T_1]}\ (\texttt{cons}\ v_M^1\ v_M^2) \to \texttt{cons}\ (SM^{T_1}\ v_M^1)\ (SM^{[T_1]}\ v_M^2)$

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

Case 4.32. ${}^{[T_1]}HM^{[T_1]}$ $({}^{[T_1]}MH^{[T_1]}$ $(\mathtt{cons}\ e_H^1\ e_H^2)) \to \mathtt{cons}\ e_H^1\ e_H^2$

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

Case 4.33. $SM^{[T_1]}$ $({}^{[T_1]}MH^{[T_1]}$ $(\mathtt{cons}\ e_H^1\ e_H^2)) \to SH^{[T_1]}$ $(\mathtt{cons}\ e_H^1\ e_H^2)$

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

Case 4.34. ${}^{[T_1]}HS$ $(SH^{[T_1]}$ $(\mathtt{cons}\ e_H^1\ e_H^2)) \to \mathtt{cons}\ e_H^1\ e_H^2$

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

Case 4.35. ${}^{[T_1]}MS$ $(SH^{[T_1]}$ $(\mathtt{cons}\ e_H^1\ e_H^2)) \to {}^{[T_1]}MH^{[T_1]}$ $(\mathtt{cons}\ e_H^1\ e_H^2)$

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

Case 4.36. ${}^{[T_1]}AS\ v_S^1 \rightarrow {}^{[T_1]}AS$ ($\texttt{wrong}$ "Not a list") *($v_S^1 \neq \texttt{cons}\ v_S^2\ v_S^3$ and $v_S^1 \neq \texttt{nil}$) 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. $\vdash_S$ $\texttt{wrong}$ "Not a list" $: TST$ by inversion. $\Gamma \vdash_A {}^{[T_1]}AS$ ($\texttt{wrong}$ "Not a list") $: [T_1]$ by inversion and uniqueness of types. $\Gamma \vdash_A {}^{[T_1]}AS$ ($\texttt{wrong}$ "Not a list") $: T$ because $[T_1] = T$.

Case 4.37. *${}^{T_1^a}AS\ (SA^{T_1^a}\ B_A) \rightarrow B_A$ 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 4.38. ${}^{T_1^a}AS\ v_S \rightarrow {}^{T_1^a}AS$ ($\texttt{wrong}$ "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. $\vdash_S$ $\texttt{wrong}$ "Parametricity violated" $: TST$ by inversion. ${}^{T_1^a}AS$ ($\texttt{wrong}$ "Parametricity violated") $: T_1^a[T_i/T_i^a]$ by inversion and uniqueness of types. ${}^{T_1^a}AS$ ($\texttt{wrong}$ "Parametricity violated") $: T$ because $T_1^a[T_i/T_i^a] = T$.

Case 4.39. *${}^{T_1 \rightarrow T_2}AB^{T_1 \rightarrow T_2}\ (\lambda x_1 : T_1.e_B) \rightarrow \lambda x_2 : T_1.({}^{T_2}AB^{T_2}\ ((\lambda x_1 : T_1.e_B)\ ({}^{T_1}BA^{T_1}\ x_2)))$ 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 4.40. ${}^{T_1\rightarrow T_2}AS\ (\lambda x_1.e_S) \rightarrow \lambda x_2 : T_1[T_i/T_i^a].({}^{T_2}AS\ ((\lambda x_1.e_S)\ (SA^{T_1}\ x_2)))$ *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 4.41. ${}^{T_1\rightarrow T_2}AS\ v_S \rightarrow {}^{T_1\rightarrow T_2}AS$ (`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.* $\vdash_S$ `wrong` "Not a function" $: TST$ *by inversion.* $\Gamma \vdash_A {}^{T_1\rightarrow T_2}AS$ (`wrong` "Not a function") $: (T_1 \rightarrow T_2)[T_i/T_i^a]$ *by inversion and uniqueness of types.* $\Gamma \vdash_A {}^{T_1\rightarrow T_2}AS$ (`wrong` "Not a function") $: T$ *because* $(T_1 \rightarrow T_2)[T_i/T_i^a] = T$.

Case 4.42. $SA^{T_1\rightarrow T_2}\ (\lambda x_1 : T_1[T_i/T_i^a].e_A) \rightarrow \lambda x_2.(SA^{T_2}\ ((\lambda x_1 : T_1[T_i/T_i^a].e_A)\ ({}^{T_1}AS\ x_2)))$ *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 4.43. ${}^{\forall X.T_1}AB^{\forall X_1.T_1}\ (\Lambda X.e_B) \rightarrow \Lambda X.({}^{T_1}AB^{T_1}\ e_B)$ *where* $(A, B) \in \{(H, M), (M, H)\}$

$\Gamma \vdash_A {}^{\forall X.T_1}AB^{\forall X.T_1}\ (\Lambda X_1.e_B) : T$ *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 4.44. ${}^{\forall X.T_1}AB^{\forall X.T_1}\ ({}^{\forall X.T_1}BS\ v_S) \rightarrow {}^{\forall X.T_1}AS\ v_S$ *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 4.45. $({}^{\forall X.T_1}AS\ v_S)\ \{T_2\} \rightarrow {}^{T_1[T_2^a/X]}AS\ v_S$ *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* $T_1[T_2^a/X][T_i/T_i^a] = T_1[T_2/X] = T$.

Case 4.46. $SA^{\forall X.T_1}\ (\Lambda X.e_A) \rightarrow SA^{T_1[L/X]}\ ((\Lambda X.e_A)\ \{L\})$ *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.*

$\square$

Chapter 4

Related Work

This work extends the work of Kinghorn [7] 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 [8] 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. [5] 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 [10] [9] [3].

Tobin-Hochstadt and Felleisen [12] 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 [6] 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 [1] 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.

Bibliography

[1] N. Benton. Embedded interpreters. *J. Funct. Program.*, 15(4):503–542, 2005.

[2] M. Blume and D. McAllester. A sound (and complete) model of contracts. *SIGPLAN Not.*, 39(9):189–200, 2004.

[3] ECMA. *Common Language Infrastructure (CLI)*, 4th edition, June 2006.

[4] R. B. Findler and M. Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, 2002.

[5] A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-parametric polymorphic contracts. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 29–40, New York, NY, USA, 2007. ACM.

[6] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: translating scheme to ml. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 192–203, New York, NY, USA, 1995. ACM.

[7] D. L. Kinghorn. Preserving parametricity while sharing higher-order, polymorphic functions between scheme and ml. Master's thesis, California Polytechnic State University, San Luis Obispo, June 2007.

[8] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *SIGPLAN Not.*, 42(1):3–10, 2007.

[9] Microsoft. *Microsoft Interface Definition Language*, November 2007.

[10] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification*, 3.1 edition, August 2004.

[11] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[12] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 964–974, New York, NY, USA, 2006. ACM.