A REWRITING LOGIC APPROACH TO DEFINING TYPE SYSTEMS

BY

CHARLES MCEWEN ELLISON III

B.S., North Carolina State University, 2004
B.S., North Carolina State University, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Adviser:

Assistant Professor Grigore Roşu

Abstract

We show how programming language semantics and definitions of their corresponding type systems can both be written in a single framework amenable to proofs of soundness. The framework is based on full rewriting logic (not to be confused with context reduction or term rewriting), where rules can match anywhere in a term (or configuration).

We present an extension of the syntactic approach to proving type system soundness presented by Wright and Felleisen [1994] that works in the above described semantics-based domain. As before, the properties of preservation and progress are crucial. We use an abstraction function to relate semantic configurations in the language domain to semantic configurations in the type domain, and then proceed to use the preservation and progress properties as usual. We also develop an abstract type system, which is a type system modulo certain structural characteristics.

To demonstrate the method, we give examples of five languages and corresponding type systems. They include two imperative languages and three functional languages, and three type checkers and two type inferencers. We then proceed to prove that preservation holds for each.

Table of Contents

List of Figures . . . . . **iv**

Chapter 1 Introduction & Comparisons **1**
1.1 Background Information 1
1.2 Statement of the Problem 3
1.3 Statement of the Solution 4
1.4 Related Work 4
1.5 Outline of Thesis 5

Chapter 2 Description of Method **6**
2.1 Proof Description 6

Chapter 3 Examples **8**
3.1 Simple Imperative Language (SIL) 8
3.2 Simple Imperative Language with Functions (SILF) 14
3.3 Monomorphic Lambda Calculus (Mono) 23
3.4 Polymorphic Lambda Calculus (Poly) 27
3.5 Exp & $\mathscr{W}$ 31

Chapter 4 Conclusions **44**
4.1 Statement of Results 44
4.2 Problems Left Unsolved 44

References **45**

List of Figures

3.1 K-style Definition of SIL 9
3.2 Expanded Definition of SIL 10
3.3 Definition of SIL's Type Checker 11
3.4 Definition of α_{SIL} 12
3.5 Definition of SILF 14
3.6 Definition of SILF's Type Checker 16
3.7 Definition of α_{SILF} 19
3.8 K-style Definition of Mono 23
3.9 Expanded Definition of Mono 23
3.10 K-style Definition of Mono's Type Checker 24
3.11 Expanded Definition of Mono's Type Checker 24
3.12 Definition of α_{Mono} 24
3.13 Definition of Poly 27
3.14 Definition of Poly's Type Inferencer 27
3.15 Definition of α_{Poly} 28
3.16 K-style Definition of Exp 31
3.17 Expanded Definition of Exp 32
3.18 K-style Definition of $\mathscr{W}$ 32
3.19 Expanded Definition of $\mathscr{W}$ 34
3.20 Milner's $\mathscr{J}$ Algorithm 35
3.21 Definition of Abstract Type Inferencer 36
3.22 Definition of α_{Exp} 37

Chapter 1

Introduction & Comparisons

1.1 Background Information

In order to understand the problem addressed in this thesis, it is necessary to understand the traditional method of proving type system soundness, as well as how our particular language design framework works. Here we introduce preliminary material necessary to understand the remainder of the paper.

1.1.1 Type System Soundness

One would often like to restrict the set of acceptable programs in a given programming language. This is done as an aid to the programmer—if a particular program is necessarily ill-formed, or even likely ill-formed, it is convenient to report this to the programmer *before* runtime, when a problem could affect an end-user. To specify what it means for a program to be ill-formed is a difficult problem, and "ill-formedness" is almost always a property beyond the capabilities of a context-free grammar. This gives rise to a kind of "static execution" provided by static type systems. They can be thought of as abstract interpreters of programs.

A type system is of particular importance when it can guarantee the absence of a certain set of errors from running programs. Thus, we say a type system is *sound* if programs that are approved by the type system are guaranteed not to encounter some particular set of errors we call "type errors." What constitutes a type error can vary by language and by type system, but generally involves calling a function with arguments that are in some way unexpected. For example, a function that returns the square of a number should never be called with a boolean value as its input, because squaring a boolean value does not make sense.

Proving type system soundness was once an ad-hoc process, varying wildly from system to system, typically needing to be reworked from scratch whenever a new feature was added to the language [Wright and Felleisen, 1994, Section 2]. A general principle of proving soundness using a context-reduction definition of a language was developed by Wright and Felleisen [1994]. This method hinges on showing two key properties:

1. If a program type checks, then after taking a step of evaluation, it still type checks.
2. If a program type checks, then during evaluation, it is either the case that you have evaluated to a value, or you can take another step of computation.

The first property is called *preservation* (or subject reduction [Curry and Feys, 1958]), and the second *progress*.

1.1.2 A Short Introduction to K

K is a framework for defining programming languages and type systems. It is important to understand its theory and syntax in order to understand the remainder of this paper. The best introduction to K can be found in Roşu [2006, 2007, 2008], but a brief (and necessarily incomplete) introduction, in order to make this document partially self-contained, is included below. It is not the goal of this thesis to explain the intricacies of K—we only present enough to aid a reader in understanding the definitions and proofs to follow.

K is based on rewriting logic. Meseguer's rewriting logic [Meseguer, 1992], not to be confused with context reduction or term rewriting, organizes term rewriting modulo equations as a logic with a complete proof system and initial model semantics. Using rewriting logic, we get a lot of machinery "for free." Not only do we get operational and algebraic denotational semantics for our languages, but we also get formal analysis tools such as a model checker and inductive theorem prover [Meseguer and Roşu, 2004, 2007]. K is similar in spirit to the Chemical Abstract Machine [Berry and Boudol, 1990], where pieces of the configuration float in a solution (an associative/commutative "soup," or multi-set) and can match with reaction rules. In K, multi-sets are also the basic elements of configurations, together with lists.

To be explicit, K is an extended subset of rewriting logic. It is a subset of rewriting logic in the sense that it suggests certain stylistic conventions to be adopted by language designers in order to implement their languages. This restriction streamlines the logic in order to offer pre-built language modules, to make definitions more consistent, and to make them more modular. Additionally, K is restricted in that it does not use any conditional rules, which are allowed in rewriting logic.[1] It is an extension of rewriting logic in the sense that the actual semantics of the logic enables more concurrency than pure rewriting logic. For more information on the concurrency of K, see Roşu [2008].

At the heart of rewriting logic are rewriting rules. Any time a rule's left-hand side (LHS) is able to match a part of the configuration, the rule applies and the subterm is transformed based on the rule. Each side is allowed to contain variables, although variables on the RHS must appear on the LHS. Rules are written in a number of general styles. *Structural* rules are internal rules used for massaging the form of the configuration. Structural rules are written $\text{LHS} = \text{RHS}$ or $\frac{\text{LHS}}{\text{RHS}}$. We use the equality symbol to suggest that configurations resulting from the application of structural rules are actually identical, or at least in the same equivalence class for the sake of any reasoning. *Semantic* rules are the rules that do actual work. Semantic rules are written $\text{LHS} \longrightarrow \text{RHS}$ or $\frac{\text{LHS}}{\text{RHS}}$. In either case, we use the following symbols to match elements from the middle, the beginning, or the end of a list or set: $\langle\!|X|\!\rangle$, $\langle\!|X\rangle\!\rangle$, $\langle X|\!\rangle$ respectively. For example, to state a rule where you replace all Xs in a set with 0, you can simply state $\frac{\langle\!|X|\!\rangle}{0}$.

Take a look at Figure 3.1 to see a simple example of a full K-definition. K-definitions are divided into three distinct parts—a description of the syntax of the language, a description of program configurations (partially evaluated programs), and a list of rewriting rules. In syntactic

[1] Indeed, conditional rules can typically be eliminated from any rewriting logic definition. See Şerbănuţă and Roşu [2006] and their references for more information.

based definitions like context rewriting, program configurations are identified with the syntax of the language. In that universe, partially evaluated terms are simply other terms. We introduce the syntax of our language using traditional grammar notation such as BNF. This syntax is then annotated in a way which helps generate rules automatically. For example, we use the idea of an operator's *strictness* to suggest which of its operands should be evaluated before the expression itself should be evaluated. These strictnesses are notated on the grammar of the language, such as in Figure 3.1a, and then used to generate rules like 6 and 7 in Figure 3.2a.

We use a few conventions in the notation of this document to shorten statements and to aid understanding. We use the $x[y \leftarrow z]$ and $x[z/y]$ notations for "z replaces y in x" interchangeably. We add $\mathcal{L}$ and $\mathcal{T}$ subscripts on constructs that are shared between both the language and the type system. We then only mention the system in which reductions are taking place if it is not immediately clear from context. A statement like $\mathcal{T} \models R \xrightarrow{*} R'$ means that R reduces to R' under the rewrite rules for $\mathcal{T}$. Types in K-rules are inferred by context, and used to do matching. If not immediately clear from context, they are made explicit. One could also take the approach of defining certain variable names to be of a particular type. We use the variables I, V, E, and K stand for integers, values, expressions, and computation items respectively, which make statements easier to understand.

We use a sans-serif font to represent syntax available to users who write programs in the languages being defined. We use an *italic* font to represent grammar types, such as *Int* being the type of any integer, or *ConfigItem* being the type of a particular configuration item. We use a SMALL-CAPS font to represent type values generated by a type inferencer or a type checker. Note that although these two are both *types*, they are very different kinds. There are many things of type *Int*, and each is a construct of syntax, while there is only a single INT, it lives in a non-syntactic world, and it used internally and emitted by type systems.

1.2 Statement of the Problem

The K-framework, a modular programming language definitional style, was introduced by Roşu [2005]. Since its inception, much work has been done on language definitions in this style [Chen et al., 2006, Hills and Roşu, 2006, 2007a,b, Meredith et al., 2007, Roşu, 2006]. However, even though many different languages have been formalized within this framework, the work on type systems of these languages has been minimal. Although type systems were given and discussed in Roşu [2006, chap. 10], the proofs of soundness were left as an exercise to the reader. This thesis attempts to provide some tools and infrastructure to assist in these kinds of proofs. It is based off a previous work by Ellison et al. [2008].

Although K uses rewriting, it uses the full power of rewriting logic [Meseguer, 1992], as opposed to the rewriting of reduction semantics [Felleisen and Hieb, 1992, Plotkin, 2004]. Because traditional preservation and progress proofs require the use of context-rewriting, this means we cannot simply adopt their proof style directly. Indeed, for any non-trivial language, the K-framework typically results in definitions that are semantics based. That is to say, configurations of partially evaluated programs are not required to be syntactically correct programs. This is in stark contrast to definitions given in a context-rewriting style, where intermediate terms are necessarily syntactically correct programs. Furthermore, K-style definitions of type systems are written in the same manner as programs. Intermediary type checking configurations, for example, are a semantic combination of

syntax and types. K-style type systems can be thought of as taking a term in the original language, and abstractly evaluating it to a type. These differences lead us to the conclusion that a new kind of proof approach is needed to address the differences of K-based languages.

Additionally, K-style definitions are usually very concrete. They are often written so as to immediately provide an interpreter, simply by executing the rewriting rules in a rewriting logic engine such as Maude [Clavel et al., 2002]. This makes many proofs about a language difficult, and so a way is needed to abstract over many of the "insignificant" features of any particular program, such as its variable names.

1.3 Statement of the Solution

Although we cannot simply adopt the proof style of Wright and Felleisen, as previously discussed, we can still use the ideas of preservation and progress. However, for the concepts to make sense, we must provide a mapping function from arbitrary language configurations, resulting from a partial evaluation of a term, to configurations in the type system domain. With this function in hand, we can prove preservation and progress and show the desired soundness properties.

To address the issue of proving properties in a framework where definitions are as concrete as an interpreter, we also develop the idea of an abstract type inferencer in Section 3.5.4. This is not a type inferencer that can generate abstract (such as polymorphic) types, but instead is a type inferencer that works modulo a number of structural properties. This lets us prove our desired properties more easily. However, it still remains an actual K definition, so any other proof techniques developed to work with K will work with an abstract K definition.

1.4 Related Work

Rewriting logic semantics is similar to other kinds of rewriting, such as context-sensitive, or context-reduction definitions, as in [Felleisen and Hieb, 1992, Matthews et al., 2004, Wright and Felleisen, 1994], but in rewriting logic, rules apply everywhere by default. In defining languages in this style, one typically tries to limit where rules can apply, whereas in context-reduction definitions, one has to specify where rewriting can happen.

Our type system definitions are similar in spirit to those independently introduced by Kuan et al. [2007] in that their techniques, like ours, involve rewriting terms to their types. However, underlying K definitions is rewriting logic, which provides both an algebraic denotation semantics, as well as a structured operational semantics [Meseguer and Roşu, 2007]. Additionally, K offers the concept of a program configuration that is a first class element of any semantics.

There has been other previous work combining rewriting logic with type systems. For example, Bravenboer et al. [2005] describes a method of using rewriting to add typecheck notations to a program. Their work consists entirely of an example typechecker, with no analysis or exposition. Also, pure type systems, which are a generalization of the λ-cube [Barendregt, 1991], have been represented in in membership equational logic [Stehr and Meseguer, 2004], a subset of rewriting logic. There was also a kind of predecessor-to-K rewriting work shown in Roşu [2003]. These notes include rewriting definitions of both languages and type systems, given in the same syntax.

There is a large body of work on term graph rewriting [Barendregt et al., 1987, Plump, 1998] and its applications to type systems [Banach, 1992, Fogarty et al., 2007, Wells, 2002]. There are similarities with our work, such as using a similar syntax for both types and terms, and a process of reduction or normalization to reduce programs to their types.

There was a type theory and term rewriting project [Kamareddine, 1997–2000] which resulted in much theoretical work, including over twenty papers on type theory and term rewriting. A collection of these papers were published together [Kamareddine and Klop, 2000]. Their goals included using "a programming language partially or entirely based on explicit concepts of term rewriting" to combine languages and their type systems [Kamareddine and Wells, 1997–2000].

Adding rewrite rules as annotations to a particular language in order to assist a separate algorithm with type checking has been explored [Huencke and de Moor, 2001], as well as adding type annotations to rewrite rules that define program transformations [Mametjanov, 2007].

Of course, much work has been done on defining type systems modularly and proving them sound [Klein and Nipkow, 2006, Lee et al., 2007, Levin and Pierce, 2003, von Oheimb and Nipkow, 1999]. Much of the recent work in mechanically verified proofs of type soundness has been stimulated by the PoplMark Challenge [Aydemir et al., 2005].

1.5 Outline of Thesis

In Chapter 2, we begin by presenting the general structure of our proof strategy. Here, we discuss what our strategy of proof looks like at the most general level. We also state the particular properties we aim to show, and what those properties look like in both our work and in previous work. Chapter 3 is the meat of this thesis, which includes five different language definitions and their corresponding type systems, together with proofs of preservation. Each language is described in at least one K formalism, necessary lemmas are stated, for which proofs are often suggested, then a number of select preservation cases are examined. We finish up in Chapter 4 by giving an analysis of the overall contribution of this thesis, as well as a description of work that still needs to be done.

Chapter 2

Description of Method

2.1 Proof Description

As discussed in Section 1.2, we need a way of correlating partially evaluated programs (configurations in the language domain) with their types. Obviously the correlated types should be related in such a way so as to preserve the type of the original language term. The way we do this is by using an abstraction function, α, which takes a configuration in the language domain to a corresponding configuration in the typing domain. Using an abstraction function to prove soundness is a technique used frequently in the domain of processor construction, as introduced in Hosabettu et al. [1998], or compiler optimization Kanade et al. [2006, 2007]. One can think of this technique as a generalization of the syntax driven approach—when your partially evaluated programs are programs themselves, then the abstraction function is just the identity function. Indeed, in many of our simple examples, α is little more than an identity function. For an example of a significantly more complicated function, see Section 3.2.3.

It is necessary to define an α for each language/type system pair as part of the proof of soundness. We think of this as a formalization of the precise relationship between a language and its type system. This relationship needs to be established in any proof of type system soundness, not just those using K—our technique simply makes this an explicit step, with the end result being an actual function from domain to domain.

We outline below the general statement and lemmas for preservation and progress in each example language and type system tried so far:

1. Preservation: If $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} V$ for some type τ and value V, then $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$.
   (a) Lemma: Structural rules preserve type. Typically, these immediately follow due to identical rules in the type system.
   (b) Main Lemma: If $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R, then $\mathcal{T} \models \alpha(R) \xrightarrow{*} \tau$.
   (c) Lemma: If $\mathcal{T} \models \alpha(V) \xrightarrow{*} \tau$ then $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$.
2. Progress: For any expression E where $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R, either R is a value, or $\exists\ R'$ such that $R \longrightarrow R'$.
   (a) Lemma: Every subterm of a well-typed term is well typed

In comparison, the definition of preservation as given by Wright and Felleisen [1994, Lemma 4.3] states: "If $\Gamma \triangleright e_1 : \tau$ and $e_1 \longrightarrow e_2$ then $\Gamma \triangleright e_2 : \tau$." We cannot define preservation in the same way,

because our terms do not necessarily remain terms as they evaluate. In fact, in non-trivial languages they end up in a configuration whose equivalence class contains no language term. If one accepts the idea of our abstraction function, then subject reduction is actually closer in spirit to the above Lemma 1b.

Although we do not focus our attention on progress in this thesis, it is important to note the proof of progress in cases where our configurations are equationally equivalent to language terms is relatively easy—it is equivalent to the same proof in a context reduction framework. However, when configurations are not identifiable with terms in the language, more work needs to be done to generalize the idea of "subterm" and how it applies to program configurations that are not themselves terms of the original language.

Chapter 3

Examples

Below we present five languages, corresponding type systems and abstraction functions, together with formal proofs of preservation. Each example is laid out in a similar manner. First, we have the K-style definition. This definition includes annotated grammars for the language and language configurations, as well as any additional evaluation rules necessary. This definition should be the most compact, and it is what is thought of as being "the" K-definition of a language. Next, optionally, we have the expanded definition, which decodes each of the annotations. Any rules that were implicit are made explicit, and each rule is numbered so referring to them is easy. Additionally, transformations may be applied such as changing the rules of a language to match only at the top of the continuation. We can do this without changing the semantics because our rules are orthogonal [Klop, 1992].[1] A similar thing is now done for the type system—a K-style definition is given, followed by an expanded definition. The final definition is that of our abstraction function α from the language domain to the type domain. Finally, we complete each section with a proof of preservation.

We have defined and proved soundness for a number of simple languages and their type systems including one for an extremely simple imperative language (SIL), a slightly more complicated imperative language with functions (SILF), as well as simply typed lambda calculus, lambda calculus with let-polymorphism (as presented in Wright and Felleisen [1994]), and Milner's Exp language and $\mathscr{W}$ algorithm [Milner, 1978].

3.1 Simple Imperative Language (SIL)

We wanted to start with a near-trivial language to do our initial proofs. We chose a simple imperative language with basic arithmetic and comparison, if statements, while statements, and assignment. Expressions can only be of two types—integer or boolean. Variables can only be assigned integers.

In this example, we give a slightly more verbose K-style definition, to help readers understand how K is organized. In most definitions, built-in types such as integers or variables do not need to be mentioned. Additionally, some configuration grammar or grammar schemata is imported and/or generated from the K-prelude, so defining things like K is unnecessary.

[1] However, although this transformation does not change the final outcome of program execution, it does decrease the amount of concurrency in the execution of a definition.

3.1.1 Language Definition

$$
\begin{array}{rcl}
\mathit{Int} & ::= & \mathbb{Z} \\
\mathit{Bool} & ::= & \mathbb{B} \\
\mathit{Var} & ::= & \text{all identifiers; to be used as names of variables} \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Var} \\
& \mid & \mathit{Exp} + \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ +_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
& \mid & \mathit{Exp} < \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ <_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
& \mid & \mathit{Exp}\ \mathsf{and}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \wedge_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
& \mid & \mathsf{not}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \neg_{\mathit{Bool}\to\mathit{Bool}}] \\
\mathit{Stmt} & ::= & \mathit{Stmt}; \mathit{Stmt}\ [\mathit{strict}(1)] \\
& \mid & \mathit{Var} := \mathit{Exp}\ [\mathit{strict}(2)] \\
& \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt}\ [\mathit{strict}(1)] \\
& \mid & \mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Stmt} \\
\mathit{Pgm} & ::= & \mathit{Stmt}; \mathit{Exp}\ [s; e = s \curvearrowright e]
\end{array}
$$

(a) Language Syntax

$$
\begin{array}{rcl}
\mathit{Val} & ::= & \mathit{Int} \mid \mathit{Bool} \\
\mathit{Result} & ::= & \mathit{Val} \\
\mathit{Store} & ::= & \mathsf{Map}[\mathit{Var}, \mathit{Val}] \\
\mathit{ConfigItem} & ::= & (\!|\mathsf{Set}[\mathit{ConfigItem}]|\!)_{\top} \mid (\!|K|\!)_{k_{\mathcal{L}}} \mid (\!|\mathit{Store}|\!)_{\mathit{store}} \\
\mathit{Config} & ::= & \mathit{Val} \mid [\![\mathsf{Set}[\mathit{ConfigItem}]]\!] \mid [\![K]\!] \\
K & ::= & \mathit{Result} \mid \mathit{KLabel}(\mathsf{List}[K]) \mid \mathsf{List}_{\curvearrowright}[K] \\
\mathit{KLabel} & ::= & \mathsf{skip} \mid \text{One for each language construct (e.g. } +, <, 1, 2, 3, \ldots)
\end{array}
$$

(b) Configuration Syntax

$$
\begin{array}{rcl}
[\![p]\!] & = & (\!|(\!|p|\!)_{k_{\mathcal{L}}}(\!|\cdot|\!)_{\mathit{store}}|\!)_{\top} \\
(\!|(\!|v|\!)_{k_{\mathcal{L}}}|\!)_{\top} & \longrightarrow & v \\
(\!|\underline{\quad x \quad}|\!)_k(\!|\sigma|\!)_{\mathit{store}} & & \sigma[x] \\
(\!|\underline{x = v}|\!)_k(\!|\underline{\quad \sigma \quad}|\!)_{\mathit{store}} & & \cdot \qquad \sigma[x \leftarrow v] \\
\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 & \longrightarrow & s_1 \\
\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 & \longrightarrow & s_2 \\
(\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)_k & = & (\!|\mathsf{if}\ b\ \mathsf{then}\ s; \mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{else\ skip}|\!)_k
\end{array}
$$

(c) Language Rules

Figure 3.1: K-style Definition of SIL

$$[\![K]\!]_{\mathcal{L}} = (\!|(\!|K|\!)_{k_{\mathcal{L}}}(\!|\cdot|\!)_{store}|\!)_{\top} \quad (1)$$

$$(\!|\underbrace{([X, I]\ \_)[X]}_{I}|\!)_{k_{\mathcal{L}}} \quad (2)$$

$$(\!|\underbrace{S[X]}_{0}|\!)_{k_{\mathcal{L}}} \text{ if } X \notin S \quad (3)$$

$$([X, \_]\ S)[X \leftarrow I] = [X, I]\ S \quad (4)$$

$$S[X \leftarrow I] = [X, I]\ S \text{ if } X \notin S \quad (5)$$

$$(\!|\underbrace{K + K'}_{K \curvearrowright \Box + K'}|\!)_{k_{\mathcal{L}}} \quad (6)$$

$$(\!|\underbrace{K + K'}_{K' \curvearrowright K + \Box}|\!)_{k_{\mathcal{L}}} \quad (7)$$

$$(\!|\underbrace{\mathsf{not}\ K}_{K \curvearrowright \mathsf{not}\ \Box}|\!)_{k_{\mathcal{L}}} \quad (8)$$

$$(\!|\underbrace{K\ \mathsf{and}\ K'}_{K \curvearrowright \Box\ \mathsf{and}\ K'}|\!)_{k_{\mathcal{L}}} \quad (9)$$

$$(\!|\underbrace{K\ \mathsf{and}\ K'}_{K' \curvearrowright K\ \mathsf{and}\ \Box}|\!)_{k_{\mathcal{L}}} \quad (10)$$

$$(\!|\underbrace{K < K'}_{K \curvearrowright \Box < K'}|\!)_{k_{\mathcal{L}}} \quad (11)$$

$$(\!|\underbrace{K < K'}_{K' \curvearrowright K < \Box}|\!)_{k_{\mathcal{L}}} \quad (12)$$

$$(\!|\underbrace{X := K}_{K \curvearrowright X := \Box}|\!)_{k_{\mathcal{L}}} \quad (13)$$

$$(\!|\underbrace{K\ ;\ K'}_{K \curvearrowright \Box\ ;\ K'}|\!)_{k_{\mathcal{L}}} \quad (14)$$

$$(\!|\underbrace{\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S'}_{K \curvearrowright \mathsf{if}\ \Box\ \mathsf{then}\ S\ \mathsf{else}\ S'}|\!)_{k_{\mathcal{L}}} \quad (15)$$

$$(\!|\underbrace{\mathsf{while}\ K\ \mathsf{do}\ K'}_{\mathsf{if}\ K\ \mathsf{then}\ (K'\ ;\ \mathsf{while}\ K\ \mathsf{do}\ K')\ \mathsf{else}\ \mathsf{skip}}|\!)_{k_{\mathcal{L}}} \quad (16)$$

(a) Structural Rules

$$(\!|(\!|V|\!)_{k_{\mathcal{L}}}|\!)_{\top} \longrightarrow V \quad (17)$$

$$(\!|\underbrace{X}_{\sigma[X]}|\!)_{k_{\mathcal{L}}}(\!|\sigma|\!)_{store} \quad (18)$$

$$(\!|\underbrace{I + I'}_{I +_{int} I'}|\!)_{k_{\mathcal{L}}} \quad (19)$$

$$(\!|\underbrace{B\ \mathsf{and}\ B'}_{B\ and_{bool}\ B'}|\!)_{k_{\mathcal{L}}} \quad (20)$$

$$(\!|\underbrace{\mathsf{not}\ B}_{not_{bool}\ B}|\!)_{k_{\mathcal{L}}} \quad (21)$$

$$(\!|\underbrace{I < I'}_{I <_{int}\ I'}|\!)_{k_{\mathcal{L}}} \quad (22)$$

$$(\!|\underbrace{\mathsf{skip}\ ;\ K}_{K}|\!)_{k_{\mathcal{L}}} \quad (23)$$

$$(\!|\underbrace{X := I}_{\cdot}|\!)_{k_{\mathcal{L}}}(\!|\underbrace{S}_{S[X \leftarrow I]}|\!)_{state} \quad (24)$$

$$(\!|\underbrace{\mathsf{if\ true\ then}\ S\ \mathsf{else}\ S'}_{S}|\!)_{k_{\mathcal{L}}} \quad (25)$$

$$(\!|\underbrace{\mathsf{if\ false\ then}\ S\ \mathsf{else}\ S'}_{S'}|\!)_{k_{\mathcal{L}}} \quad (26)$$

(b) Semantic Rules

Figure 3.2: Expanded Definition of SIL

3.1.2 Type Checker Definition

Notice that in this definition of a type checker for SIL, we need to include a rule for typing skip. This is, in essence, an artifact of needing the capability of typing arbitrary language configurations. We discussed a number of possible alternatives to this not-so-elegant solution. One is to simply dissolve all statements in the type checker. This would be paired with changes to the language semantics that dissolve all statements. However, options like this are not explored in this thesis.

$$[\![E]\!]_{\mathcal{T}} = (\!|(\!|E|\!)_{k_{\mathcal{T}}}|\!)_{\top} \quad (27)$$

$$\frac{(\!| \quad K + K' \quad |\!)_{k_{\mathcal{T}}}}{K \curvearrowright \square + K'} \quad (28)$$

$$\frac{(\!| \quad K + K' \quad |\!)_{k_{\mathcal{T}}}}{K' \curvearrowright K + \square} \quad (29)$$

$$\frac{(\!| \quad \mathsf{not}\ K \quad |\!)_{k_{\mathcal{T}}}}{K \curvearrowright \mathsf{not}\ \square} \quad (30)$$

$$\frac{(\!| \quad K\ \mathsf{and}\ K' \quad |\!)_{k_{\mathcal{T}}}}{K \curvearrowright \square\ \mathsf{and}\ K'} \quad (31)$$

$$\frac{(\!| \quad K\ \mathsf{and}\ K' \quad |\!)_{k_{\mathcal{T}}}}{K' \curvearrowright K\ \mathsf{and}\ \square} \quad (32)$$

$$\frac{(\!| \quad K < K' \quad |\!)_{k_{\mathcal{T}}}}{K \curvearrowright \square < K'} \quad (33)$$

$$\frac{(\!| \quad K < K' \quad |\!)_{k_{\mathcal{T}}}}{K' \curvearrowright K < \square} \quad (34)$$

$$\frac{(\!| \quad X := K \quad |\!)_{k_{\mathcal{T}}}}{K \curvearrowright X := \square} \quad (35)$$

$$\frac{(\!| \quad K\ ;\ K' \quad |\!)_{k_{\mathcal{T}}}}{K \curvearrowright \square\ ;\ K'} \quad (36)$$

$$\frac{(\!| \quad \mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' \quad |\!)_{k_{\mathcal{T}}}}{K \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ S\ \mathsf{else}\ S'} \quad (37)$$

$$\frac{(\!| \quad \mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' \quad |\!)_{k_{\mathcal{T}}}}{S \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ \square\ \mathsf{else}\ S'} \quad (38)$$

$$\frac{(\!| \quad \mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' \quad |\!)_{k_{\mathcal{T}}}}{S' \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ \square} \quad (39)$$

$$\frac{(\!| \quad \mathsf{while}\ K\ \mathsf{do}\ K' \quad |\!)_{k_{\mathcal{T}}}}{K \curvearrowright \mathsf{while}\ \square\ \mathsf{do}\ K'} \quad (40)$$

$$\frac{(\!| \quad \mathsf{while}\ K\ \mathsf{do}\ K' \quad |\!)_{k_{\mathcal{T}}}}{K' \curvearrowright \mathsf{while}\ K\ \mathsf{do}\ \square} \quad (41)$$

(a) Structural Rules

$$(\!|(\!|\tau|\!)_{k_{\mathcal{T}}}|\!)_{\top} \longrightarrow \tau \quad (42)$$

$$V : Int \longrightarrow \textsc{Int} \quad (43)$$

$$B : Bool \longrightarrow \textsc{Bool} \quad (44)$$

$$\mathsf{skip} \longrightarrow \textsc{Statement} \quad (45)$$

$$S[X] \longrightarrow \textsc{Int} \quad (46)$$

$$\textsc{Int} + \textsc{Int} \longrightarrow \textsc{Int} \quad (47)$$

$$\textsc{Bool}\ \mathsf{and}\ \textsc{Bool} \longrightarrow \textsc{Bool} \quad (48)$$

$$\mathsf{not}\ \textsc{Bool} \longrightarrow \textsc{Bool} \quad (49)$$

$$\textsc{Int} < \textsc{Int} \longrightarrow \textsc{Bool} \quad (50)$$

$$\textsc{Statement}\ ;\ K \longrightarrow K \quad (51)$$

$$X := \textsc{Int} \longrightarrow \textsc{Statement} \quad (52)$$

$$\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \tau\ \mathsf{else}\ \tau \longrightarrow \tau \quad (53)$$

$$\mathsf{while}\ \textsc{Bool}\ \mathsf{do}\ \textsc{Statement} \longrightarrow \textsc{Statement} \quad (54)$$

(b) Semantic Rules

Figure 3.3: Definition of SIL's Type Checker

3.1.3 Abstraction Function Definition

This abstraction function is little more than the identity function. It can apply on either completely unevaluated, partially evaluated, or fully evaluated programs, translating them into corresponding configurations in the type domain.

$$\alpha([\![E]\!]_{\mathcal{L}}) = [\![E]\!]_{\mathcal{T}} \tag{55}$$
$$\alpha((\!(\!(K)\!)_{k_{\mathcal{L}}}(\!(\_)\!)_{store})\top) = (\!(\!(K)\!)_{k_{\mathcal{T}}})\top \tag{56}$$
$$\alpha(K) = [\![K]\!]_{\mathcal{T}} \tag{57}$$

Figure 3.4: Definition of α_{SIL}

3.1.4 Proofs

Lemma 1. *If* $\mathcal{T} \models \alpha(V) \xrightarrow{*} \tau$ *then* $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$

Proof. This follows directly from rule 57. □

Lemma 2 (Top of Stack Lemma)**.** *If* $(\!(S \curvearrowright K)\!)_{k_{\mathcal{T}}} \xrightarrow{*} (\!(\tau \curvearrowright K)\!)_{k_{\mathcal{T}}}$*, then* $(\!(S \curvearrowright K')\!)_{k_{\mathcal{T}}} \xrightarrow{*} (\!(\tau \curvearrowright K)\!)_{k_{\mathcal{T}}}$ *for any* K'*.*

Lemma 3 (Main Lemma for Preservation)**.** *Let* E *be an expression such that* $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ *and* $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ *for some* τ *and* R*. Then* $\mathcal{T} \models \alpha(R) \xrightarrow{*} \tau$*.*

Proof. The proof proceeds by induction on the number of steps taken to get from $[\![E]\!]_{\mathcal{L}}$ to R.

Base Case Assume no steps were taken. $\alpha(R) = \alpha([\![E]\!]_{\mathcal{L}})$ and we define $\alpha([\![E]\!]_{\mathcal{L}})$ as $[\![E]\!]_{\mathcal{T}}$ by rule 55, so by assumption we have $\alpha(R) = \tau$.

Induction case Assume $[\![E]\!]_{\mathcal{L}} \xrightarrow{n} R$ and $\mathcal{T} \models \alpha(R) \xrightarrow{*} \tau$. If no steps can be taken from R then the property holds vacuously, so assume an $n+1$ step can be taken to get to a state R'. This step could be any one of the rules of the language. We consider each individually.

Rules 6 through 15 These are taken care of automatically by the fact that each has a corresponding structural rule in the type system.

Rule 16 $R = (\!(\!(\mathsf{while}\ K\ \mathsf{do}\ K' \curvearrowright J)\!)_{k_{\mathcal{L}}}(\!(\_)\!)_{store})\top$

$\alpha(R)$	By Rule
$(\!(\mathsf{while}\ K\ \mathsf{do}\ K' \curvearrowright J)\!)_{k_{\mathcal{T}}}$	56
$(\!(\textsc{Statement} \curvearrowright J)\!)_{k_{\mathcal{T}}}$	ind. assm. & 54

$R' = (\!(\!(\mathsf{if}\ K\ \mathsf{then}\ (K'\,;\mathsf{while}\ K\ \mathsf{do}\ K')\ \mathsf{else\ skip} \curvearrowright J)\!)_{k_{\mathcal{L}}}(\!(\_)\!)_{store})\top$

$\alpha(R')$	By Rule
$(\!(\mathsf{if}\ K\ \mathsf{then}\ (K'\,;\mathsf{while}\ K\ \mathsf{do}\ K')\ \mathsf{else\ skip} \curvearrowright J)\!)_{k_{\mathcal{T}}}$	56
$(\!(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ (\textsc{Statement}\,;\mathsf{while}\ \textsc{Bool}\ \mathsf{do}\ \textsc{Statement})\ \mathsf{else\ skip} \curvearrowright J)\!)_{k_{\mathcal{T}}}$	ind. assm. & 54
$(\!(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ (\textsc{Statement}\,;\textsc{Statement})\ \mathsf{else\ skip} \curvearrowright J)\!)_{k_{\mathcal{T}}}$	54
$(\!(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \textsc{Statement}\ \mathsf{else\ skip} \curvearrowright J)\!)_{k_{\mathcal{T}}}$	51
$(\!(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \textsc{Statement}\ \mathsf{else}\ \textsc{Statement} \curvearrowright J)\!)_{k_{\mathcal{T}}}$	45
$(\!(\textsc{Statement} \curvearrowright J)\!)_{k_{\mathcal{T}}}$	53

Rule 17 $R = (\!(\!(V)\!)_{k_{\mathcal{L}}}(\!(\_)\!)_{store})\!)_{\top}$

$\alpha(R)$	By Rule
$(\!(\!(V)\!)_{k_{\mathcal{T}}})\!)_{\top}$	56
$[\![V]\!]_{\mathcal{T}}$	27

$R' = V$

$\alpha(R')$	By Rule
$[\![V]\!]_{\mathcal{T}}$	57

Rule 19 $R = (\!(\!(I : Int + I' : Int \curvearrowright K)\!)_{k_{\mathcal{L}}}(\!(\_)\!)_{store})\!)_{\top}$

$\alpha(R)$	By Rule
$(\!(I + I' \curvearrowright K)\!)_{k_{\mathcal{T}}}$	56
$(\!(\textsc{Int} + I' \curvearrowright K)\!)_{k_{\mathcal{T}}}$	43
$(\!(\textsc{Int} + \textsc{Int} \curvearrowright K)\!)_{k_{\mathcal{T}}}$	43
$(\!(\textsc{Int} \curvearrowright K)\!)_{k_{\mathcal{T}}}$	47

$R' = (\!(\!(I +_{int} I' \curvearrowright K)\!)_{k_{\mathcal{L}}}(\!(\_)\!)_{store})\!)_{\top}$

$\alpha(R')$	By Rule
$(\!(I +_{int} I' \curvearrowright K)\!)_{k_{\mathcal{T}}}$	56
$(\!(\textsc{Int} \curvearrowright K)\!)_{k_{\mathcal{T}}}$	43

Rule 24 $R = (\!(\!(X := I : Int \curvearrowright K)\!)_{k_{\mathcal{L}}}(\!(S)\!)_{store})\!)_{\top}$

$\alpha(R)$	By Rule
$(\!(X := I \curvearrowright K)\!)_{k_{\mathcal{T}}}$	56
$(\!(X := \textsc{Int} \curvearrowright K)\!)_{k_{\mathcal{T}}}$	43
$(\!(\textsc{Statement} \curvearrowright K)\!)_{k_{\mathcal{T}}}$	52

$R' = (\!(\!(\mathsf{skip} \curvearrowright K)\!)_{k_{\mathcal{L}}}(\!(S[V \leftarrow I])\!)_{store})\!)_{\top}$

$\alpha(R')$	By Rule
$(\!(\mathsf{skip} \curvearrowright K)\!)_{k_{\mathcal{T}}}$	56
$(\!(\textsc{Statement} \curvearrowright K)\!)_{k_{\mathcal{T}}}$	45

Rule 25 $R = (\!(\!(\mathsf{if\ true\ then}\ S\ \mathsf{else}\ S' \curvearrowright K)\!)_{k_{\mathcal{L}}}(\!(\_)\!)_{store})\!)_{\top}$

$\alpha(R)$	By Rule
$(\!(\mathsf{if\ true\ then}\ S\ \mathsf{else}\ S' \curvearrowright K)\!)_{k_{\mathcal{T}}}$	56
$(\!(\mathsf{if\ true\ then}\ \tau\ \mathsf{else}\ S' \curvearrowright K)\!)_{k_{\mathcal{T}}}$	ind. assm.
$(\!(\mathsf{if\ true\ then}\ \tau\ \mathsf{else}\ \tau \curvearrowright K)\!)_{k_{\mathcal{T}}}$	ind. assm.
$(\!(\tau \curvearrowright K)\!)_{k_{\mathcal{T}}}$	53

$R' = (\!(\!(S \curvearrowright K)\!)_{k_{\mathcal{L}}}(\!(\_)\!)_{store})\!)_{\top}$

$\alpha(R')$	By Rule
$(\!(S \curvearrowright K)\!)_{k_{\mathcal{T}}}$	56
$(\!(\tau \curvearrowright K)\!)_{k_{\mathcal{T}}}$	ind. assm. & Lemma 2

□

Theorem 1 (Preservation)**.** *If* $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ *and* $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} V$ *for some type* τ *and value* V*, then* $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$

Proof. This follows directly from Lemmas 1 and 3. □

3.2 Simple Imperative Language with Functions (SILF)

SILF originated in Hills et al. [2007], and was further developed in Roşu [2008]. The below version is actually a pared down version of the language presented in those papers, but still contains function calls, which is the most interesting part for our purposes.

3.2.1 Language Definition

$$
\begin{array}{rcl}
\mathit{Int} & ::= & \mathbb{Z} \\
\mathit{Bool} & ::= & \mathbb{B} \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables and functions} \\
\mathit{Type} & ::= & \mathit{int} \mid \mathit{bool} \ \text{(one may add more types if one extends the language)} \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \\
 & \mid & \mathit{Name} \\
 & \mid & \mathsf{read}() \\
 & \mid & \mathit{Name}(\mathsf{List}[\mathit{Exp}])\ [\mathit{strict}(1),\quad f(\mathit{el}_1, e, \mathit{el}_2) = e \curvearrowright f(\mathit{el}_1, \square, \mathit{el}_2)] \\
 & \mid & \mathit{Exp} + \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ +_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} < \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ <_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{and}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \wedge_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathsf{not}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \neg_{\mathit{Bool}\to\mathit{Bool}}] \\
\mathit{Decl} & ::= & \mathsf{var}\ \mathit{Type}\ \mathit{Name} \mid \mathsf{var}\ \mathit{Type}\ \mathit{Name}[\mathit{Nat}] \\
 & \mid & \mathit{Decl}; \mathit{Decl}\ [d_1; d_2 = d_1 \curvearrowright d_2] \\
\mathit{Stmt} & ::= & \{\}\ [\{\} = \cdot] \\
 & \mid & \{\mathit{Stmt}\}\ [\{s\} = s] \\
 & \mid & \{\mathit{Decl}; \mathit{Stmt}\} \\
 & \mid & \mathit{Stmt}; \mathit{Stmt}\ [s_1; s_2 = s_1 \curvearrowright s_2] \\
 & \mid & \mathsf{write}(\mathit{Exp})\ [\mathit{strict}] \\
 & \mid & \mathit{Name} = \mathit{Exp}\ [\mathit{strict}(2)] \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt}\ [\mathit{strict}(1)] \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ [\mathsf{if}\, e\ \mathsf{then}\, s = \mathsf{if}\, e\ \mathsf{then}\, s\ \mathsf{else}\, \cdot] \\
 & \mid & \mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Stmt} \\
 & \mid & \mathsf{call}\ \mathit{Exp}\ [\mathit{strict}] \\
 & \mid & \mathsf{return}\ \mathit{Exp}\ [\mathit{strict}] \\
\mathit{FunDecl} & ::= & \mathsf{function}\ \mathit{Type}\ \mathit{Name}(\mathsf{List}[\mathit{Type}]\ \mathsf{List}[\mathit{Name}])\ \mathit{Stmt} \\
 & \mid & \mathit{FunDecl}\, \mathit{FunDecl}\ [\mathit{fd}_1\ \mathit{fd}_2 = \mathit{fd}_1 \curvearrowright \mathit{fd}_2] \\
\mathit{Pgm} & ::= & \mathit{FunDecl} \\
 & \mid & \mathit{Decl}; \mathit{FunDecl}\ [d; \mathit{fd} = d \curvearrowright \mathit{fd}]
\end{array}
$$

(a) Language Syntax

Figure 3.5: Definition of SILF

$$
\begin{array}{rcl}
\mathit{Val} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Type} \mid \lambda \mathsf{List}[\mathit{Type}]\ \mathsf{List}[\mathit{Name}].K \\
\mathit{Result} & ::= & \mathit{Val} \\
\mathit{Env} & ::= & \mathsf{Map}[\mathit{Name}, \mathit{Loc} \times \mathit{Type}]\ (\forall \rho \in \mathit{Env}, x \in \mathit{Name}, \text{let } \rho[x] \text{ be } (\mathit{loc}(\rho[x]), \mathit{type}(\rho[x]))) \\
\mathit{Store} & ::= & \mathsf{Map}[\mathit{Loc}, \mathit{Val}] \\
\mathit{FStack} & ::= & \mathsf{List}[\mathit{Env} \times K \times \mathit{Type}] \\
\mathit{ConfigItem} & ::= & (\!|K|\!)_{k_\mathcal{L}} \mid (\!|\mathit{FStack}|\!)_{\mathit{fstack}} \mid (\!|\mathit{Env}|\!)_{\mathit{env}_\mathcal{L}} \mid (\!|\mathit{Env}|\!)_{\mathit{genv}_\mathcal{L}} \\
& \mid & (\!|\mathsf{List}[\mathit{Int}]|\!)_{\mathit{in}} \mid (\!|\mathsf{List}[\mathit{Int}]|\!)_{\mathit{out}} \mid (\!|\mathit{Type}|\!)_{\mathit{return}_\mathcal{L}} \mid (\!|\mathit{Store}|\!)_{\mathit{store}} \mid (\!|\mathit{Loc}|\!)_{\mathit{nextLoc}} \\
\mathit{Config} & ::= & \mathsf{List}[\mathit{Int}] \mid [\![K, \mathsf{List}[\mathit{Int}]]\!]_\mathcal{L} \mid (\!|\mathsf{Set}[\mathit{ConfigItem}]|\!)_\top \\
& \mid & [\![K]\!]_\mathcal{L}\ [[\![k]\!]_\mathcal{L} = [\![k, \cdot]\!]_\mathcal{L}] \\
K & ::= & \ldots \mid \mathit{run} \mid \mathit{restore}(\mathit{Env}) \\
\mathit{Type} & ::= & \ldots \mid ? \mid \mathit{Type}[\mathit{Nat}]
\end{array}
$$

(b) Configuration Syntax

$$(\!|p, il|\!)_\top = (\!|(\!|p \curvearrowright \mathit{run}|\!)_{k_\mathcal{L}}\ (\!|\cdot|\!)_{\mathit{fstack}}\ (\!|\cdot|\!)_{\mathit{env}_\mathcal{L}}\ (\!|\cdot|\!)_{\mathit{genv}_\mathcal{L}}\ (\!|il|\!)_{\mathit{in}}\ (\!|\cdot|\!)_{\mathit{out}}\ (\!|?|\!)_{\mathit{return}_\mathcal{L}}\ (\!|\cdot|\!)_{\mathit{store}}\ (\!|\mathit{loc}(0)|\!)_{\mathit{nextLoc}}|\!)_\top \quad (58)$$

$$\langle\!|(\!|\cdot|\!)_{k_\mathcal{L}}\ (\!|il|\!)_{\mathit{out}}|\!\rangle_\top = il \quad (59)$$

$$(\!|\underset{\text{call main}}{\underbrace{\mathit{run}}}|\!)_{k_\mathcal{L}}\ (\!|\rho|\!)_{\mathit{env}_\mathcal{L}}\ (\!|\underset{\rho}{\underbrace{\cdot}}|\!)_{\mathit{genv}_\mathcal{L}} \quad (60)$$

$$(\!|\underset{\sigma[\mathit{loc}(\rho[x])]}{\underbrace{x}}|\!)_{k_\mathcal{L}}\ (\!|\rho|\!)_{\mathit{env}_\mathcal{L}}\ (\!|\sigma|\!)_{\mathit{store}} \quad (61)$$

$$(\!|\underset{i}{\underbrace{\mathsf{read}()}}|\!)_{k_\mathcal{L}}\ (\!|\underset{\cdot}{\underbrace{i}}|\!)_{\mathit{in}} \quad (62)$$

$$(\!|\underset{s}{\underbrace{(t\ \lambda tl\, xl.s)(vl) \curvearrowright k}}|\!)_{k_\mathcal{L}}\ (\!|\underset{(\rho, k, t')}{\underbrace{\cdot}}|\!)_{\mathit{fstack}}\ (\!|\underset{\rho'[xl \leftarrow (ll', tl)]}{\underbrace{\rho}}|\!)_{\mathit{env}_\mathcal{L}}\ (\!|\rho'|\!)_{\mathit{genv}_\mathcal{L}}\ (\!|\underset{t}{\underbrace{t'}}|\!)_{\mathit{return}_\mathcal{L}}$$
$$(\!|\underset{\sigma[ll' \leftarrow vl]}{\underbrace{\sigma}}|\!)_{\mathit{store}}\ (\!|\underset{l'}{\underbrace{l}}|\!)_{\mathit{nextLoc}} \text{ where } l' \text{ is } l +_{\mathit{Loc}} |xl|,\ ll' \text{ is } l \ldots (l-1), \text{ and } \mathit{typeOf}(vl){=}tl \quad (63)$$

$$(\!|\underset{\cdot}{\underbrace{\mathsf{var}\ t\ x}}|\!)_{k_\mathcal{L}}\ (\!|\underset{\rho[x \leftarrow (l, t)]}{\underbrace{\rho}}|\!)_{\mathit{env}_\mathcal{L}}\ (\!|\underset{l +_{\mathit{Loc}} 1}{\underbrace{l}}|\!)_{\mathit{nextLoc}} \quad (64)$$

$$(\!|\underset{d \curvearrowright s \curvearrowright \mathit{restore}(\rho)}{\underbrace{\{d; s\}}}|\!)_{k_\mathcal{L}}\ (\!|\rho|\!)_{\mathit{env}_\mathcal{L}} \quad (65)$$

$$(\!|\underset{\cdot}{\underbrace{\mathit{restore}(\rho)}}|\!)_{k_\mathcal{L}}\ (\!|\underset{\rho}{\underbrace{\_}}|\!)_{\mathit{env}_\mathcal{L}} \quad (66)$$

$$(\!|\underset{\cdot}{\underbrace{\mathsf{write}\ i}}|\!)_{k_\mathcal{L}}\ \langle\!|\underset{i}{\underbrace{\cdot}}|\!\rangle_{\mathit{out}} \quad (67)$$

$$(\!|\underset{\cdot}{\underbrace{x = v}}|\!)_{k_\mathcal{L}}\ (\!|\rho|\!)_{\mathit{env}_\mathcal{L}}\ (\!|\underset{\sigma[\mathit{loc}(\rho[x]) \leftarrow v]}{\underbrace{\sigma}}|\!)_{\mathit{store}} \quad \text{where } \mathit{type}(\rho[x]) = \mathit{typeOf}(v) \quad (68)$$

$$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \longrightarrow s_1 \quad (69)$$

$$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \longrightarrow s_2 \quad (70)$$

$$(\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)_{k_\mathcal{L}} = (\!|\mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)|\!)_{k_\mathcal{L}} \quad (71)$$

$$\mathsf{call}\ v \longrightarrow . \quad (72)$$

$$(\!|\underset{v \curvearrowright k}{\underbrace{\mathsf{return}\ v \curvearrowright \_}}|\!)_{k_\mathcal{L}}\ (\!|\underset{\cdot}{\underbrace{(\rho, k, t)}}|\!)_{\mathit{fstack}}\ (\!|\underset{\rho}{\underbrace{\_}}|\!)_{\mathit{env}_\mathcal{L}}\ (\!|\underset{t}{\underbrace{t'}}|\!)_{\mathit{return}_\mathcal{L}} \quad \text{if } \mathit{typeOf}(v) = t' \quad (73)$$

$$(\!|\underset{\cdot}{\underbrace{\mathsf{function}\ t\ f(tl\ xl)\ s}}|\!)_{k_\mathcal{L}}\ (\!|\underset{\rho[f \leftarrow (l, ?)]}{\underbrace{\rho}}|\!)_{\mathit{env}_\mathcal{L}}\ (\!|\underset{\sigma[l \leftarrow t\ \lambda tl\ xl.s]}{\underbrace{\sigma}}|\!)_{\mathit{store}}\ (\!|\underset{l +_{\mathit{Loc}} 1}{\underbrace{l}}|\!)_{\mathit{nextLoc}} \quad (74)$$

(c) Language Rules

Figure 3.5: Definition of SILF (cont.)

3.2.2 Type Checker Definition

It is important to note that except for the annotations, the syntax for SILF's type checker is identical to that of the language itself. This is true for all K-style language/type system pairs.

$$
\begin{array}{rcl}
\mathit{Int} & ::= & \mathbb{Z} \\
\mathit{Bool} & ::= & \mathbb{B} \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables and functions} \\
\mathit{Type} & ::= & \textsc{Int} \mid \textsc{Bool} \quad \text{(one may add more types if one extends the language)} \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \\
& \mid & \mathit{Name} \\
& \mid & \mathsf{read}() \\
& \mid & \mathit{Name}(\mathsf{List}[\mathit{Exp}])\ [\mathit{strict}(1), \quad f(el_1, e, el_2) = e \curvearrowright f(el_1, \square, el_2)] \\
& \mid & \mathit{Exp} + \mathit{Exp}\ [\mathit{strict}] \\
& \mid & \mathit{Exp} < \mathit{Exp}\ [\mathit{strict}] \\
& \mid & \mathit{Exp}\ \mathsf{and}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid & \mathsf{not}\ \mathit{Exp}\ [\mathit{strict}] \\
\mathit{Decl} & ::= & \mathsf{var}\ \mathit{Type}\ \mathit{Name} \mid \mathsf{var}\ \mathit{Type}\ \mathit{Name}[\mathit{Nat}] \\
& \mid & \mathit{Decl}; \mathit{Decl}\ [\mathit{strict}] \\
\mathit{Stmt} & ::= & \{\} \\
& \mid & \{\mathit{Stmt}\}\ [\mathit{strict}] \\
& \mid & \{\mathit{Decl}; \mathit{Stmt}\} \\
& \mid & \mathit{Stmt}; \mathit{Stmt}\ [\mathit{strict}] \\
& \mid & \mathsf{write}(\mathit{Exp})\ [\mathit{strict}] \\
& \mid & \mathit{Name} = \mathit{Exp}\ [\mathit{strict}] \\
& \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt}\ [\mathit{strict}] \\
& \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ [\mathit{strict}] \\
& \mid & \mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Stmt}\ [\mathit{strict}] \\
& \mid & \mathsf{call}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid & \mathsf{return}\ \mathit{Exp}\ [\mathit{strict}] \\
\mathit{FunDecl} & ::= & \mathsf{function}\ \mathit{Type}\ \mathit{Name}(\mathsf{List}[\mathit{Type}]\ \mathsf{List}[\mathit{Name}])\ \mathit{Stmt} \\
& \mid & \mathit{FunDecl}\,\mathit{FunDecl}\ [\mathit{strict}] \\
\mathit{Pgm} & ::= & \mathit{FunDecl} \\
& \mid & \mathit{Decl}; \mathit{FunDecl}\ [\mathit{strict}]
\end{array}
$$

(a) Language Syntax

$$
\begin{array}{rcl}
\mathit{Type} & ::= & \ldots \mid ? \mid \textsc{Decl} \mid \textsc{Statement} \mid \textsc{FunDecl} \mid \textsc{Program} \mid \mathsf{List}[\mathit{Type}] \to \mathit{Type} \\
\mathit{Result} & ::= & \mathit{Type} \\
\mathit{TEnv} & ::= & \mathsf{Map}[\mathit{Name}, \mathit{Type}] \\
\mathit{ConfigItem} & ::= & (\!|K|\!)_{k_\mathcal{T}} \mid (\!|\mathit{TEnv}|\!)_{env_\mathcal{T}} \mid (\!|\mathit{TEnv}|\!)_{genv_\mathcal{T}} \mid (\!|\mathit{Type}|\!)_{return_\mathcal{T}} \mid (\!|\cdot|\!)_{to\,Type} \\
\mathit{Config} & ::= & \mathit{done} \mid [\![K]\!]_\mathcal{T} \mid (\!|\mathsf{Set}[\mathit{ConfigItem}]|\!)_\top \\
K & ::= & \ldots \mid \mathit{restore}(\mathit{TEnv}) \mid \mathit{Type}\ \lambda\mathsf{List}[\mathit{Type}]\,\mathsf{List}[\mathit{Name}].K
\end{array}
$$

(b) Configuration Syntax

Figure 3.6: Definition of SILF's Type Checker

$$(\!|p|\!)\top = (\!|(\!|p|\!)_{k_\mathcal{T}}\ (\!|\cdot|\!)_{env_\mathcal{T}}\ (\!|\cdot|\!)_{genv_\mathcal{T}}\ (\!|?|\!)_{return_\mathcal{T}}\ (\!|\cdot|\!)_{toType}|\!)\top \tag{75}$$

$$(\!|t \curvearrowright \underset{\cdot}{\underset{\rho}{restore(\rho)}}|\!)_{k_\mathcal{T}}\ (\!|\_|\!)_{env_\mathcal{T}} \tag{76}$$

(c) Structural Rules

$$i \longrightarrow \textsc{Int} \tag{77}$$

$$b \longrightarrow \textsc{Bool} \tag{78}$$

$$\mathsf{read}() \longrightarrow \textsc{Int} \tag{79}$$

$$(tl \to t)(tl) \longrightarrow t \tag{80}$$

$$\textsc{Int} + \textsc{Int} \longrightarrow \textsc{Int} \tag{81}$$

$$\textsc{Int} < \textsc{Int} \longrightarrow \textsc{Bool} \tag{82}$$

$$\textsc{Bool}\ \mathsf{and}\ \textsc{Bool} \longrightarrow \textsc{Bool} \tag{83}$$

$$\textsc{Bool}\ \mathsf{or}\ \textsc{Bool} \longrightarrow \textsc{Bool} \tag{84}$$

$$\mathsf{not}\ \textsc{Bool} \longrightarrow \textsc{Bool} \tag{85}$$

$$\textsc{Decl}; \textsc{Decl} \longrightarrow \textsc{Decl} \tag{86}$$

$$\{\} \longrightarrow \textsc{Statement} \tag{87}$$

$$\{\textsc{Statement}\} \longrightarrow \textsc{Statement} \tag{88}$$

$$\textsc{Decl}; \textsc{Statement} \longrightarrow \textsc{Statement} \tag{89}$$

$$\mathsf{write}\ \textsc{Int} \longrightarrow \textsc{Statement} \tag{90}$$

$$(t = t) \longrightarrow \textsc{Statement} \tag{91}$$

$$\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \textsc{Statement}\ \mathsf{else}\ \textsc{Statement} \longrightarrow \textsc{Statement} \tag{92}$$

$$\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \textsc{Statement} \longrightarrow \textsc{Statement} \tag{93}$$

$$\mathsf{while}\ \textsc{Bool}\ \mathsf{do}\ \textsc{Statement} \longrightarrow \textsc{Statement} \tag{94}$$

$$\mathsf{call}\ t \longrightarrow \textsc{Statement} \tag{95}$$

$$\textsc{FunDecl}\ \textsc{FunDecl} \longrightarrow \textsc{FunDecl} \tag{96}$$

$$\textsc{Decl}; \textsc{FunDecl} \longrightarrow \textsc{Program} \tag{97}$$

(d) Semantic Rules

Figure 3.6: Definition of SILF's Type Checker (cont.)

$$\langle\!(\cdot)\!\rangle_{k_\mathcal{T}}\ (\!|\cdot|\!)_{toType}\rangle\!\rangle_{\top} = done \tag{98}$$

$$(\!|\underline{\ x\ }|\!)_{k_\mathcal{T}}\ (\!|\rho|\!)_{env_\mathcal{T}} \tag{99}$$
$$\rho[x]$$

$$(\!|\underline{\mathsf{var}\ t\ x}|\!)_{k_\mathcal{T}}\ (\!|\underline{\quad\rho\quad}|\!)_{env_\mathcal{T}} \tag{100}$$
$$\textsc{Decl}\qquad\rho[x \leftarrow t]$$

$$(\!|\underline{\quad\{d; s\}\quad}|\!)_{k_\mathcal{T}}\ (\!|\rho|\!)_{env_\mathcal{T}} \tag{101}$$
$$d; s \curvearrowright restore(\rho)$$

$$(\!|\underline{\mathsf{return}\ t \curvearrowright \_}|\!)_{k_\mathcal{T}}\ (\!|\underline{t}|\!)_{return_\mathcal{T}} \tag{102}$$
$$\cdot\qquad\qquad ?$$

$$(\!|\underline{\mathsf{function}\ t\ f(tl\ xl)\ s}|\!\rangle_{k_\mathcal{T}}\ (\!|\underline{\quad\rho\quad}|\!)_{env_\mathcal{T}}\ \langle\!|\underline{\quad\cdot\quad}|\!\rangle_{toType} \tag{103}$$
$$\textsc{FunDecl}\qquad\rho[f \leftarrow (tl \to t)]\qquad t\ \lambda tl\, xl.s$$

$$(\!|\underline{\textsc{Program}}|\!)_{k_\mathcal{T}}\ (\!|\rho|\!)_{env_\mathcal{T}}\ (\!|\underline{\cdot}|\!)_{genv_\mathcal{T}} \tag{104}$$
$$\cdot\qquad\qquad\rho$$

$$(\!|\underline{\textsc{FunDecl}}|\!)_{k_\mathcal{T}}\ (\!|\rho|\!)_{env_\mathcal{T}}\ (\!|\underline{\cdot}|\!)_{genv_\mathcal{T}} \tag{105}$$
$$\cdot\qquad\qquad\rho$$

$$(\!|\underline{\cdot}|\!)_{k_\mathcal{T}}\ (\!|\underline{\quad\rho\quad}|\!)_{env_\mathcal{T}}\ (\!|\rho'|\!)_{genv_\mathcal{T}}\ (\!|\underline{t'}|\!)_{return_\mathcal{T}}\ (\!|\underline{t\ \lambda tl\ xl.s}|\!\rangle_{toType} \tag{106}$$
$$s\qquad\rho'[xl \leftarrow tl]\qquad\qquad t\qquad\qquad\cdot$$

(e) Semantic Rules, Continued

Figure 3.6: Definition of SILF's Type Checker (cont.)

3.2.3 Abstraction Function Definition

We want to define a function that will take a configuration in the statically type-checked language domain and convert it to a corresponding configuration in the static type-checker domain.

Observations:

1. *run* remains on the stack until the entire global environment is computed
2. While the global environment is being computed, only variable and function declarations are on the stack
3. After the global environment is computed, it is never changed again
4. Saving function declarations is atomic
5. In the type system, everything after a return is discarded

With these observations in mind, we now proceed to define α.

α:

$$\alpha([\![p, il]\!]_{\mathcal{L}}) = [\![p]\!]_{\mathcal{T}} \tag{107}$$

$$\alpha(il) = done \tag{108}$$

$$\begin{aligned}&\alpha(\langle\!(\langle\!(p)\!\rangle_{k_{\mathcal{L}}}\ \langle\!(\Lambda)\!\rangle_{fstack}\ \langle\!(\rho)\!\rangle_{env_{\mathcal{L}}}\ \langle\!(\omega)\!\rangle_{genv_{\mathcal{L}}}\ \langle\!(il)\!\rangle_{in}\ \langle\!(il')\!\rangle_{out}\ \langle\!(\tau)\!\rangle_{return_{\mathcal{L}}}\ \langle\!(\sigma)\!\rangle_{store}\ \langle\!(l)\!\rangle_{nextLoc}\rangle_{\top})\\ &= \langle\!(\langle\!(\alpha_k(p))\!\rangle_{k_{\mathcal{T}}}\ \langle\!(\alpha_{env}(\rho,\sigma))\!\rangle_{env_{\mathcal{T}}}\ \langle\!(\alpha_{env}(\omega,\sigma))\!\rangle_{genv_{\mathcal{T}}}\ \langle\!(\tau)\!\rangle_{return_{\mathcal{T}}}\ \langle\!(\alpha_{toType}(\sigma))\!\rangle_{toType}\rangle_{\top}\end{aligned} \tag{109}$$

α_k:

$$\alpha_k(p) = \cdot \qquad \text{if } run \notin p \tag{110}$$

$$\alpha_k(p) = p \qquad \text{if } run \in p \text{ but } p \neq run \tag{111}$$

$$\alpha_k(p) = \textsc{FunDecl} \qquad \text{if } p = run \tag{112}$$

α_{env}:

$$\alpha_{env}(\langle\!|x \leftarrow (l,\tau)|\!\rangle, \langle\!|l \leftarrow v|\!\rangle) = (x \leftarrow \alpha_{vl}(v,\tau))\ \alpha_{env}(\langle\!|\cdot|\!\rangle, \langle\!|\cdot|\!\rangle) \tag{113}$$

$$\alpha_{env}(\langle\!|\cdot|\!\rangle, \sigma) = \cdot \tag{114}$$

α_{vl}:

$$\alpha_{vl}(v,\tau) = \tau \quad \text{if } \tau \neq ? \tag{115}$$

$$\alpha_{vl}(\tau\ \lambda\pi l\ xl\ .\ s, \_) = \pi l \to \tau \tag{116}$$

α_{toType}:

$$\alpha_{toType}(\langle\!|l \leftarrow (\tau\ \lambda\pi l\ xl\ .\ s)|\!\rangle) = (\tau\ \lambda\pi l\ xl\ .\ s)\ \alpha_{toType}(\langle\!|\cdot|\!\rangle)\alpha_{toType}(\sigma) \tag{117}$$

$$= \cdot \quad \text{otherwise} \tag{118}$$

Figure 3.7: Definition of α_{SILF}

3.2.4 Proofs

When we state what preservation means in SILF, we run into an interesting thing. Our type checker is indiscriminate—it only says whether programs pass through, not what particular type they are. Thus, we can consider all final values (which in SILF are lists of values) to be passing programs. We see this in our definition of α of value lists. According to rule 108, all value lists are automatically "done." Therefore, our secondary preservation lemma is very unnatural. Indeed, it belies the fact that what we call the main lemma for preservation is the best analogue for preservation in our setting, despite the fact that it contains α directly.

Lemma 4. $\forall$ *reachable language configurations* C, $\forall X \in$ *env of* C, $loc(X) \in$ *store of* C.

Lemma 5. $\forall$ *reachable language configurations* C, $\forall X \in$ *genv of* C, $loc(X) \in$ *store of* C.

Lemma 6. $\forall$ *reachable language configurations* C, *let* ρ *be the env of* C *and* σ *be the store of* C. $\alpha_{env}(\rho, \sigma) \xrightarrow{*}$ *some* $\mathsf{Map}[Name, Type]$.

Lemma 7. $\forall$ *reachable language configurations* C, *let* ω *be the genv of* C *and* σ *be the store of* C. $\alpha_{env}(\omega, \sigma) \xrightarrow{*}$ *some* $\mathsf{Map}[Name, Type]$.

Lemma 8. $\forall$ *reachable language configurations* C, *let* σ *be the store of* C. $\alpha_{toType}(\sigma) \xrightarrow{*}$ *some* $\mathsf{List}[Type\ \lambda\mathsf{List}[Type]\ \mathsf{List}[Name]]$.

Theorem 2 (Preservation)**.** *If* $[\![p]\!]_{\mathcal{T}} \xrightarrow{*}$ *done and* $[\![p]\!]_{\mathcal{L}} \xrightarrow{*} R$ *for some* R, *then* $\mathcal{T} \models \alpha(R) \xrightarrow{*}$ *done.*

Proof. The proof proceeds by induction on the number of steps taken to get from $[\![p]\!]_{\mathcal{L}}$ to R.

Base Case Assume no steps were taken. Then $R = [\![p]\!]_{\mathcal{L}}$. We see that $\alpha(R) = \alpha([\![p]\!]_{\mathcal{L}}) = \alpha([\![p, \cdot]\!]_{\mathcal{L}})$, so by α-rule 107, we see that $\alpha(R) = [\![p]\!]_{\mathcal{T}}$. By assumption, this reduces to *done*, so we have that $\alpha(R) \xrightarrow{*}$ *done*.

Induction Case Assume $[\![p]\!]_{\mathcal{L}} \xrightarrow{n} R$ and $\alpha(R) \xrightarrow{*}$ *done*. We want to show that if R' is a language configuration such that $\mathcal{L} \models R \longrightarrow R'$, then $\alpha(R') \xrightarrow{*}$ *done*. The step from R to R' could be any one of the structural or semantic rules of the language. We consider each individually:

Rule 59 $R = \langle\!(\langle\!(\cdot)\!\rangle_{k_{\mathcal{L}}}\ (\!|il|\!)_{out})\!\rangle_{\top}$, $R' = il$
$\alpha(R)$:

$$\alpha(R) = \alpha(\langle\!(\langle\!(\cdot)\!\rangle_{k_{\mathcal{L}}}\ (\!|il|\!)_{out})\!\rangle_{\top})$$

$= done$ by ind. assm.

$\alpha(R')$:

$$\alpha(R') = \alpha(il)$$

$= done$ by α-rule 108

Rule 60 $R = \langle\!\langle \llbracket run \rrbracket_{k_{\mathcal{L}}}\ \llbracket \rho \rrbracket_{env_{\mathcal{L}}}\ \llbracket \cdot \rrbracket_{genv_{\mathcal{L}}} \rangle\!\rangle_{\top}$, $R' = \langle\!\langle \llbracket \mathsf{call\ main}() \rrbracket_{k_{\mathcal{L}}}\ \llbracket \rho \rrbracket_{env_{\mathcal{L}}}\ \llbracket \rho \rrbracket_{genv_{\mathcal{L}}} \rangle\!\rangle_{\top}$
$\alpha(R)$:

$$\alpha(R) = \alpha(\langle\!\langle \llbracket run \rrbracket_{k_{\mathcal{L}}}\ \llbracket \rho \rrbracket_{env_{\mathcal{L}}}\ \llbracket \cdot \rrbracket_{genv_{\mathcal{L}}}\ \llbracket \_ \rrbracket_{in}\ \llbracket \_ \rrbracket_{out}\ \llbracket \tau \rrbracket_{return_{\mathcal{L}}}\ \llbracket \sigma \rrbracket_{store}\ \llbracket \_ \rrbracket_{nextLoc} \rangle\!\rangle_{\top})$$

$= \langle\!\langle \llbracket \alpha_k(run, \rho) \rrbracket_{k_{\mathcal{T}}}\ \llbracket \alpha_{env}(\rho, \sigma) \rrbracket_{env_{\mathcal{T}}}\ \llbracket \alpha_{env}(\cdot, \sigma) \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket \alpha_{toType}(\sigma) \rrbracket_{toType} \rangle\!\rangle_{\top}$ by α-rule 107

$= \langle\!\langle \llbracket \textsc{FunDecl} \rrbracket_{k_{\mathcal{T}}}\ \llbracket \alpha_{env}(\rho, \sigma) \rrbracket_{env_{\mathcal{T}}}\ \llbracket \alpha_{env}(\cdot, \sigma) \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket \alpha_{toType}(\sigma) \rrbracket_{toType} \rangle\!\rangle_{\top}$ by α-rule 112

$= \langle\!\langle \llbracket \textsc{FunDecl} \rrbracket_{k_{\mathcal{T}}}\ \llbracket \rho' \rrbracket_{env_{\mathcal{T}}}\ \llbracket \alpha_{env}(\cdot, \sigma) \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket \alpha_{toType}(\sigma) \rrbracket_{toType} \rangle\!\rangle_{\top}$ by Lemma 6

$= \langle\!\langle \llbracket \textsc{FunDecl} \rrbracket_{k_{\mathcal{T}}}\ \llbracket \rho' \rrbracket_{env_{\mathcal{T}}}\ \llbracket \cdot \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket \alpha_{toType}(\sigma) \rrbracket_{toType} \rangle\!\rangle_{\top}$ by α-rule 114

$= \langle\!\langle \llbracket \textsc{FunDecl} \rrbracket_{k_{\mathcal{T}}}\ \llbracket \rho' \rrbracket_{env_{\mathcal{T}}}\ \llbracket \cdot \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket fl \rrbracket_{toType} \rangle\!\rangle_{\top}$ by Lemma 8

$= \langle\!\langle \llbracket \cdot \rrbracket_{k_{\mathcal{T}}}\ \llbracket \rho' \rrbracket_{env_{\mathcal{T}}}\ \llbracket \rho' \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket fl \rrbracket_{toType} \rangle\!\rangle_{\top}$ by typing-rule

$= done$ by ind. assm.

$\alpha(R')$:

$$\alpha(R') = \alpha(\langle\!\langle \llbracket \mathsf{call\ main}() \rrbracket_{k_{\mathcal{L}}}\ \llbracket \rho \rrbracket_{env_{\mathcal{L}}}\ \llbracket \rho \rrbracket_{genv_{\mathcal{L}}}\ \llbracket \_ \rrbracket_{in}\ \llbracket \_ \rrbracket_{out}\ \llbracket \tau \rrbracket_{return_{\mathcal{L}}}\ \llbracket \sigma \rrbracket_{store}\ \llbracket \_ \rrbracket_{nextLoc} \rangle\!\rangle_{\top})$$

$= \langle\!\langle \llbracket \alpha_k(\mathsf{call\ main}(), \rho) \rrbracket_{k_{\mathcal{T}}}\ \llbracket \alpha_{env}(\rho, \sigma) \rrbracket_{env_{\mathcal{T}}}\ \llbracket \alpha_{env}(\rho, \sigma) \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket \alpha_{toType}(\sigma) \rrbracket_{toType} \rangle\!\rangle_{\top}$ by α-rule 107

$= \langle\!\langle \llbracket \cdot \rrbracket_{k_{\mathcal{T}}}\ \llbracket \alpha_{env}(\rho, \sigma) \rrbracket_{env_{\mathcal{T}}}\ \llbracket \alpha_{env}(\rho, \sigma) \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket \alpha_{toType}(\sigma) \rrbracket_{toType} \rangle\!\rangle_{\top}$ by α-rule 110

$= \langle\!\langle \llbracket \cdot \rrbracket_{k_{\mathcal{T}}}\ \llbracket \rho' \rrbracket_{env_{\mathcal{T}}}\ \llbracket \rho' \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket \alpha_{toType}(\sigma) \rrbracket_{toType} \rangle\!\rangle_{\top}$ by Lemma 6

$= \langle\!\langle \llbracket \cdot \rrbracket_{k_{\mathcal{T}}}\ \llbracket \rho' \rrbracket_{env_{\mathcal{T}}}\ \llbracket \rho' \rrbracket_{genv_{\mathcal{T}}}\ \llbracket \tau \rrbracket_{return_{\mathcal{T}}}\ \llbracket fl \rrbracket_{toType} \rangle\!\rangle_{\top}$ by Lemma 8

Rule 74 $R = \langle\!(\!(\mathsf{function}\ t\ f(\tau l\ xl)\ s \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!|\rho|\!)_{env_{\mathcal{L}}}\ (\!|\sigma|\!)_{store}\ (\!|l|\!)_{nextLoc}\rangle\!\rangle_{\top}$,
$R' = \langle\!(\!(K)\!)_{k_{\mathcal{L}}}\ (\!|\rho[f \leftarrow (l, ?)]|\!)_{env_{\mathcal{L}}}\ (\!|\sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s]|\!)_{store}\ (\!|l +_{Loc} 1|\!)_{nextLoc}\rangle\!\rangle_{\top}$
$\alpha(R)$:

$$\alpha(R) = \alpha(\langle\!(\!(\mathsf{function}\ t\ f(\tau l\ xl)\ s \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!|\rho|\!)_{env_{\mathcal{L}}}\ (\!|\sigma|\!)_{store}\ (\!|l|\!)_{nextLoc}\rangle\!\rangle_{\top})$$

$= \langle\!(\!(\alpha_k(\mathsf{function}\ t\ f(\tau l\ xl)\ s \curvearrowright K))\!)_{k_{\mathcal{T}}}\ (\!|\alpha_{env}(\rho, \sigma)|\!)_{env_{\mathcal{T}}}\ (\!|\alpha_{env}(\cdot, \sigma)|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|\alpha_{toType}(\sigma)|\!)_{toType}\rangle\!\rangle_{\top}$ by α-rule 107

$= \langle\!(\!(\mathsf{function}\ t\ f(\tau l\ xl)\ s \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!|\alpha_{env}(\rho, \sigma)|\!)_{env_{\mathcal{T}}}\ (\!|\alpha_{env}(\cdot, \sigma)|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|\alpha_{toType}(\sigma)|\!)_{toType}\rangle\!\rangle_{\top}$ by α-rule 112

$= \langle\!(\!(\mathsf{function}\ t\ f(\tau l\ xl)\ s \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!|\rho'|\!)_{env_{\mathcal{T}}}\ (\!|\alpha_{env}(\cdot, \sigma)|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|\alpha_{toType}(\sigma)|\!)_{toType}\rangle\!\rangle_{\top}$ by Lemma 6

$= \langle\!(\!(\mathsf{function}\ t\ f(\tau l\ xl)\ s \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!|\rho'|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|\alpha_{toType}(\sigma)|\!)_{toType}\rangle\!\rangle_{\top}$ by Lemma α-rule 114

$= \langle\!(\!(\mathsf{function}\ t\ f(\tau l\ xl)\ s \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!|\rho'|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|fl|\!)_{toType}\rangle\!\rangle_{\top}$ by Lemma 8

$= \langle\!(\!(\textsc{FunDecl} \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!|\rho'[f \leftarrow (\tau l \rightarrow t)]|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|fl \cdot t\ \lambda\tau l\ xl\ .\ s|\!)_{toType}\rangle\!\rangle_{\top}$ by typing rule

$= done$ by inductive assumption

$\alpha(R')$:

$$\alpha(R') = \alpha(\langle\!(\!(K)\!)_{k_{\mathcal{L}}}\ (\!|\rho[f \leftarrow (l, ?)]|\!)_{env_{\mathcal{L}}}\ (\!|\sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s]|\!)_{store}\ (\!|l +_{Loc} 1|\!)_{nextLoc}\rangle\!\rangle_{\top})$$

$= \langle\!(\!(\alpha_k(K))\!)_{k_{\mathcal{T}}}\ (\!|\alpha_{env}(\rho[f \leftarrow (l, ?)], \sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s])|\!)_{env_{\mathcal{T}}}\ (\!|\alpha_{env}(\cdot, \sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s])|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|\alpha_{toType}(\sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s])|\!)_{toType}\rangle\!\rangle_{\top}$ by α-rule 107

$= \langle\!(\!(K)\!)_{k_{\mathcal{T}}}\ (\!|\alpha_{env}(\rho[f \leftarrow (l, ?)], \sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s])|\!)_{env_{\mathcal{T}}}\ (\!|\alpha_{env}(\cdot, \sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s])|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|\alpha_{toType}(\sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s])|\!)_{toType}\rangle\!\rangle_{\top}$ by α-rule 112

$= \langle\!(\!(K)\!)_{k_{\mathcal{T}}}\ (\!|\rho'[f \leftarrow (\tau l \rightarrow t)]|\!)_{env_{\mathcal{T}}}\ (\!|\alpha_{env}(\cdot, \sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s])|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|\alpha_{toType}(\sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s])|\!)_{toType}\rangle\!\rangle_{\top}$ by Lem. 6 & α-rule 113

$= \langle\!(\!(K)\!)_{k_{\mathcal{T}}}\ (\!|\rho'[f \leftarrow (\tau l \rightarrow t)]|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|\alpha_{toType}(\sigma[l \leftarrow t\ \lambda\tau l\ xl\ .\ s])|\!)_{toType}\rangle\!\rangle_{\top}$ by α-rule 114

$= \langle\!(\!(K)\!)_{k_{\mathcal{T}}}\ (\!|\rho'[f \leftarrow (\tau l \rightarrow t)]|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{genv_{\mathcal{T}}}\ (\!|\tau|\!)_{return_{\mathcal{T}}}\ (\!|fl \cdot (t\ \lambda\tau l\ xl\ .\ s)|\!)_{toType}\rangle\!\rangle_{\top}$ by α-rule 114

□

3.3 Monomorphic Lambda Calculus (Mono)

Here we look at an extremely simple version of lambda calculus. It is simply-typed, call-by-value, and monomorphic. For brevity, we simply call it "Mono."

3.3.1 Language Definition

$$\begin{array}{rcl} \mathit{Type} & ::= & \bullet \mid \mathit{Type} \to \mathit{Type} \\ \mathit{Exp} & ::= & \lambda\, \mathit{Var} \colon \mathit{Type}\,.\, \mathit{Exp} \mid \mathit{Exp}\ \mathit{Exp}\ [\mathit{strict}] \end{array}$$

(a) Language Syntax

$$\begin{array}{rcl} \mathit{Val} & ::= & \lambda\, \mathit{Var} \colon \mathit{Type}\,.\, \mathit{Exp} \mid \mathit{Var} \\ \mathit{Result} & ::= & \mathit{Val} \\ \mathit{ConfigItem} & ::= & (\!|K|\!)_{k_{\mathcal{L}}} \\ \mathit{Config} & ::= & \mathit{Result} \mid [\![K]\!]_{\mathcal{L}}\ [[\![k]\!]_{\mathcal{L}} = (\!|(\!|k|\!)_{k_{\mathcal{L}}}|\!)_{\top}] \mid (\!|\mathsf{Set}[\mathit{ConfigItem}]|\!)_{\top} \end{array}$$

(b) Configuration Syntax

$$\langle\!|(\!|v|\!)_{k_{\mathcal{L}}}|\!\rangle_{\top} \longrightarrow v$$
$$(\!|(\lambda x : \tau\,.\, e)\, v|\!\rangle_{k_{\mathcal{L}}} \longrightarrow (\!|e[x \leftarrow v]|\!\rangle_{k_{\mathcal{L}}}$$

(c) Language Rules

Figure 3.8: K-style Definition of Mono

$$[\![E]\!]_{\mathcal{L}} = (\!|(\!|E|\!)_{k_{\mathcal{L}}}|\!)_{\top} \quad (119)$$
$$K\,K' = K \curvearrowright \Box K' \quad (120)$$
$$K\,K' = K' \curvearrowright K\Box \quad (121)$$

(a) Structural Rules

$$\langle\!|(\!|V|\!)_{k_{\mathcal{L}}}|\!\rangle_{\top} \longrightarrow V \quad (122)$$
$$\frac{(\!|(\lambda X : \tau\,.\, E)(V : \mathit{Val})|\!\rangle_{k_{\mathcal{L}}}}{E[V/X]} \quad (123)$$

(b) Semantic Rules

Figure 3.9: Expanded Definition of Mono

3.3.2 Type Checker Definition

$$\begin{aligned} \mathit{Type} &::= \bullet \mid \mathit{Type} \to \mathit{Type}\ [\mathit{strict}(2)] \\ \mathit{Exp} &::= \lambda\, \mathit{Var} \colon \mathit{Type}\,.\,\mathit{Exp} \mid \mathit{Exp}\ \mathit{Exp} \end{aligned}$$

(a) Language Syntax

$$\begin{aligned} \mathit{Val} &::= \mathit{Type} \\ \mathit{Result} &::= \mathit{Val} \\ \mathit{Exp} &::= \mathit{Val} \mid \mathit{Exp}\ \mathit{Exp}\ [\mathit{strict}] \\ \mathit{ConfigItem} &::= (\!|K|\!)_{k_{\mathcal{T}}} \\ \mathit{Config} &::= \mathit{Result} \mid [\![K]\!]_{\mathcal{T}}\ [[\![k]\!]_{\mathcal{T}} = (\!|(\!|k|\!)_{k_{\mathcal{T}}}|\!)_{\top}] \mid (\!|\mathsf{Set}[\mathit{ConfigItem}]|\!)_{\top} \end{aligned}$$

(b) Configuration Syntax

$$\begin{aligned} \langle\!|(\!|\tau|\!)_{k_{\mathcal{L}}}|\!\rangle_{\top} &\longrightarrow \tau \\ \lambda x\colon\tau\,.\,e &\longrightarrow \tau \to e[x \leftarrow \tau] \\ (\tau \to \tau')\tau &\longrightarrow \tau' \end{aligned}$$

(c) Language Rules

Figure 3.10: K-style Definition of Mono's Type Checker

$$[\![E]\!]_{\mathcal{T}} = (\!|(\!|E|\!)_{k_{\mathcal{T}}}|\!)_{\top} \quad (124)$$

$$K\,K' = K \curvearrowright \Box K' \quad (125)$$

$$K\,K' = K' \curvearrowright K\Box \quad (126)$$

$$\tau \to K = K \curvearrowright \tau \to \Box \quad (127)$$

(a) Structural Rules

$$\langle\!|(\!|\tau|\!)_{k_{\mathcal{T}}}|\!\rangle_{\top} \longrightarrow \tau \quad (128)$$

$$\lambda X\colon\tau\,.\,K \longrightarrow \tau \to K[\tau/X] \quad (129)$$

$$(\tau \to \tau')\tau \longrightarrow \tau' \quad (130)$$

(b) Semantic Rules

Figure 3.11: Expanded Definition of Mono's Type Checker

3.3.3 Abstraction Function Definition

$$\alpha([\![E]\!]_{\mathcal{L}}) = [\![E]\!]_{\mathcal{T}} \quad (131)$$

$$\alpha((\!|(\!|K|\!)_{k_{\mathcal{L}}}|\!)_{\top}) = (\!|(\!|K|\!)_{k_{\mathcal{T}}}|\!)_{\top} \quad (132)$$

$$\alpha(V : \mathit{Val}) = [\![V]\!]_{\mathcal{T}} \quad (133)$$

Figure 3.12: Definition of α_{Mono}

3.3.4 Proofs

Lemma 9. *α is a total function over all reachable configurations of the language, modulo the structural rules of the language.*

Lemma 10. *Any reachable configuration of the language is equivalent under the structural rules to exactly one configuration of the form $[\![E]\!]_{\mathcal{L}}$ for some expression E.*

Lemma 11. *If $(\!(\!(E[\tau'/X] \curvearrowright K)\!)_{k_{\mathcal{T}}})_\top \xrightarrow{*} (\!(\!(\tau \curvearrowright K)\!)_{k_{\mathcal{T}}})_\top$ and $(\!(\!(V \curvearrowright K')\!)_{k_{\mathcal{T}}})_\top \xrightarrow{*} (\!(\!(\tau' \curvearrowright K')\!)_{k_{\mathcal{T}}})_\top$, then $(\!(\!(E[V/X] \curvearrowright K)\!)_{k_{\mathcal{T}}})_\top \xrightarrow{*} (\!(\!(\tau \curvearrowright K)\!)_{k_{\mathcal{T}}})_\top$.*

Proof. Assume $(\!(\!(E[\tau'/X] \curvearrowright K)\!)_{k_{\mathcal{T}}})_\top \xrightarrow{*} (\!(\!(\tau \curvearrowright K)\!)_{k_{\mathcal{T}}})_\top$ and $(\!(\!(V \curvearrowright K')\!)_{k_{\mathcal{T}}})_\top \xrightarrow{*} (\!(\!(\tau' \curvearrowright K')\!)_{k_{\mathcal{T}}})_\top$ for some τ', X, τ and V. We will do an induction on the size of E.

Base Case: Consider expressions E of size 0. These include only variable names. If $E = X$ then $(\!|E[\tau'/X] \curvearrowright K|\!)_{k_{\mathcal{T}}} = (\!|\tau' \curvearrowright K|\!)_{k_{\mathcal{T}}}$ and $(\!|E[V/X] \curvearrowright K|\!)_{k_{\mathcal{T}}} = (\!|V \curvearrowright K|\!)_{k_{\mathcal{T}}}$ which by assumption reduces to $(\!|\tau' \curvearrowright K|\!)_{k_{\mathcal{T}}}$. We see that the property holds in this case.

Alternatively, if $E = Y$ where $Y \neq X$ then $(\!|E[\tau'/X] \curvearrowright K|\!)_{k_{\mathcal{T}}} = (\!|Y \curvearrowright K|\!)_{k_{\mathcal{T}}}$. However, this cannot reduce to any type because we cannot type single variables. Therefore, this case cannot occur under the assumptions.

Inductive Case: Assume the above property holds for all expressions up to size n. Consider an expression of size $n+1$. It is either a lambda expression or an application expression.

If it is a lambda expression, we see $(\!|E[\tau'/X] \curvearrowright K|\!)_{k_{\mathcal{T}}} = (\!|(\lambda Y\!:\!\sigma\,.\,E')[\tau'/X] \curvearrowright K|\!)_{k_{\mathcal{T}}}$. Using the typing rule for lambda expressions, and properties of substitution, we see that this reduces to $(\!|\sigma \to (E'[\sigma/Y])[\tau'/X] \curvearrowright K|\!)_{k_{\mathcal{T}}}$. Finally, by the structural rule, we see that this is equivalent to $(\!|(E'[\sigma/Y])[\tau'/X] \curvearrowright \sigma \to \square \curvearrowright K|\!)_{k_{\mathcal{T}}}$.

Similarly, we see that $(\!|E[V/X] \curvearrowright K|\!)_{k_{\mathcal{T}}} = (\!|(\lambda Y\!:\!\sigma\,.\,E')[V/X] \curvearrowright K|\!)_{k_{\mathcal{T}}}$ for some σ and E'. Using the typing rule for lambda expressions, and properties of substitution, we see that this reduces to $(\!|\sigma \to (E'[\sigma/Y])[V/X] \curvearrowright K|\!)_{k_{\mathcal{T}}} = (\!|(E'[\sigma/Y])[V/X] \curvearrowright \sigma \to \square \curvearrowright K|\!)_{k_{\mathcal{T}}}$. Because we must be able to type this, we now see by inductive assumption that this is the same as $(\!|(E'[\sigma/Y])[\tau'/X] \curvearrowright \sigma \to \square \curvearrowright K|\!)_{k_{\mathcal{T}}}$, which is also the same as above, so the property holds in this case.

Similarly for the other cases.

□

Lemma 12. *If $\mathcal{T} \models \alpha(V) \xrightarrow{*} \tau$ then $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$*

Proof. This follows directly from rule 133. □

Lemma 13 (Main Lemma for Preservation)**.** *Let E be an expression such that $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R. Then $\mathcal{T} \models \alpha(R) \xrightarrow{*} \tau$.*

Proof. The proof proceeds by induction on the number of steps taken to get from $[\![E]\!]_{\mathcal{L}}$ to R.

Base Case Assume no steps were taken. Then $R = [\![E]\!]_{\mathcal{L}}$. By 131 we see that $\alpha(R) = [\![E]\!]_{\mathcal{T}}$. By assumption, this reduces to τ, so we have that $\alpha(R) \xrightarrow{*} \tau$.

Induction case Assume $[\![E]\!]_{\mathcal{L}} \xrightarrow{n} R$ and $\alpha(R) \xrightarrow{*} \tau$. If no steps can be taken from R then the property holds vacuously, so assume an $n+1$ step can be taken to get to a state R'. This step could be any one of the structural or semantic rules of the language. We consider each individually.

Rule 119 $R = [\![E]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![E]\!]_{\mathcal{T}}$	131

$R' = (\!(E)\!)_{k_{\mathcal{L}}})_{\top}$

$\alpha(R')$	By Rule
$(\!(E)\!)_{k_{\mathcal{T}}})_{\top}$	132
$[\![E]\!]_{\mathcal{T}}$	124

Rule 122 $R = (\!(V)\!)_{k_{\mathcal{L}}})_{\top}$

$\alpha(R)$	By Rule
$(\!(V)\!)_{k_{\mathcal{T}}})_{\top}$	132

$R' = V$

$\alpha(R')$	By Rule
$[\![V]\!]_{\mathcal{T}}$	133
$(\!(V)\!)_{k_{\mathcal{T}}})_{\top}$	128

Rules 120 and 121 These follow because the type system has corresponding structural rules 125 and 126.

Rule 123 $R = (\!(\lambda X{:}\tau' \,.\, E)(V : \mathit{Val}) \curvearrowright K)\!)_{k_{\mathcal{L}}})_{\top}$

$\alpha(R)$	By Rule
$(\!(\lambda X{:}\tau' \,.\, E)(V : \mathit{Val}) \curvearrowright K)\!)_{k_{\mathcal{T}}})_{\top}$	132
$(\!(\tau' \to E[\tau'/X])(V : \mathit{Val}) \curvearrowright K)\!)_{k_{\mathcal{T}}})_{\top}$	129
$(\!(\tau' \to E[\tau'/X])(\tau') \curvearrowright K)\!)_{k_{\mathcal{T}}})_{\top}$	ind. assm.
$(\!(E[\tau'/X] \curvearrowright K)\!)_{k_{\mathcal{T}}})_{\top}$	130

$R' = (\!(E[V/X] \curvearrowright K)\!)_{k_{\mathcal{L}}})_{\top}$

$\alpha(R')$	By Rule
$(\!(E[V/X] \curvearrowright K)\!)_{k_{\mathcal{T}}})_{\top}$	132
$(\!(E[\tau'/X] \curvearrowright K)\!)_{k_{\mathcal{T}}})_{\top}$	Above & Lemma 11

□

Theorem 3 (Preservation)**.** *If* $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ *and* $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} V$ *for some type* τ *and value* V*, then* $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$

Proof. This follows directly from Lemmas 12 and 13. □

3.4 Polymorphic Lambda Calculus (Poly)

In this section we present the full polymorphic lambda calculus with constants as described in Wright and Felleisen [1994, Section 4]. For brevity, we simply call it "Poly." We take advantage of the correspondence between their type system and one where let bindings are substituted. This change vastly simplifies our proof because it causes the type system to be more like the language semantics.

3.4.1 Language Definition

$$[\![E]\!]_{\mathcal{L}} = (\!|(\!|E|\!)_{k_{\mathcal{L}}}|\!)_{\top} \tag{134}$$

$$EE' = E \curvearrowright \Box E' \tag{135}$$

$$(V : Val)E = E \curvearrowright V\Box \tag{136}$$

$$\mathsf{let}\ X\ \mathsf{be}\ E\ \mathsf{in}\ E' = E \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \Box\ \mathsf{in}\ E' \tag{137}$$

(a) Structural Rules

$$\langle\!|(\!|V|\!)_{k_{\mathcal{L}}}|\!\rangle_{\top} \longrightarrow V \tag{138}$$

$$(\!|\underline{(\lambda X\,.\,E)(V : Val)}|\!)_{k_{\mathcal{L}}} \quad E[V/X] \tag{139}$$

$$(\!|\underline{\mathsf{let}\ X\ \mathsf{be}\ V : Val\ \mathsf{in}\ E}|\!)_{k_{\mathcal{L}}} \quad E[V/X] \tag{140}$$

$$(\!|\underline{(C : Const)(V : Val)}|\!)_{k_{\mathcal{L}}} \quad \delta(C, V) \tag{141}$$

$$(\!|\underline{\quad \Upsilon(V : Val) \quad}|\!)_{k_{\mathcal{L}}} \quad V(\lambda X\,.\,(\Upsilon V)X) \tag{142}$$

(b) Semantic Rules

Figure 3.13: Definition of Poly

3.4.2 Type Inferencer Definition

$$(\tau = \tau) = \cdot \tag{143}$$

$$\frac{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'}{(\tau_1 = \tau_1'), (\tau_2 = \tau_2')} \tag{144}$$

$$(\!|\underline{\quad \tau_e \quad}|\!)_{k_{\mathcal{T}}} \curvearrowright solve\rangle\!|_{k_{\mathcal{T}}}\ (\!|\underline{(\tau_v = \tau) \cdot \mathcal{E}}|\!)_{eqns} \Leftarrow \tau_v \notin vars(\tau) \tag{145}$$

$$(\!|\tau \curvearrowright \underline{solve}|\!\rangle_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns} \quad \cdot \tag{146}$$

(a) Unification Rules

Figure 3.14: Definition of Poly's Type Inferencer

$$[\![E]\!]_{\mathcal{T}} = (\!|(\!|E \curvearrowright solve|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}|\!)_{\top} \tag{147}$$
$$K\,K' = K \curvearrowright \square K' \tag{148}$$
$$K\,K' = K' \curvearrowright K\square \tag{149}$$
$$\tau \to K = K \curvearrowright \tau \to \square \tag{150}$$

(b) Structural Rules

$$\langle\!|(\!|\tau|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}|\!|_{\top} \longrightarrow \tau \tag{151}$$
$$(\!|\frac{\lambda X\,.\,E}{\tau_v \to E[\tau_v/X]}|\!)_{k_{\mathcal{T}}}\ (\!|\frac{\tau_v}{next(\tau_v)}|\!)_{nextType} \tag{152}$$
$$(\!|\frac{\tau\ \tau'}{\tau_v}|\!)_{k_{\mathcal{T}}}\ (\!|\frac{\cdot}{\tau = \tau' \to \tau_v}|\!)_{eqns}\ (\!|\frac{\tau_v}{next(\tau_v)}|\!)_{nextType} \tag{153}$$
$$(\!|\frac{\mathsf{let}\ X\ \mathsf{be}\ E\ \mathsf{in}\ E'}{E'[E/X]}|\!)_{k_{\mathcal{T}}} \tag{154}$$
$$(\!|\frac{C}{instantiate(\,TypeOf\,(C))}|\!)_{k_{\mathcal{T}}} \tag{155}$$
$$(\!|\frac{instantiate(\forall(\cdot).\tau)}{\tau}|\!)_{k_{\mathcal{T}}} \tag{156}$$
$$(\!|instantiate(\forall\underset{\cdot}{\langle\tau_v\rangle}.\frac{\tau}{\tau\,[\tau_v'/\tau_v]})|\!)_{k_{\mathcal{T}}}\ (\!|\frac{\tau_v'}{next(\tau_v')}|\!)_{nextType} \tag{157}$$
$$(\!|\frac{\mathsf{Y}}{((\tau_v \to next(\tau_v)) \to \tau_v \to next(\tau_v)) \to \tau_v \to next(\tau_v)}|\!)_{k_{\mathcal{T}}}\ (\!|\frac{\tau_v}{next(next(\tau_v))}|\!)_{nextType} \tag{158}$$

(c) Semantic Rules

Figure 3.14: Definition of Poly's Type Inferencer (cont.)

3.4.3 Abstraction Function Definition

$$\alpha([\![E]\!]_{\mathcal{L}}) = [\![E]\!]_{\mathcal{T}} \tag{159}$$
$$\alpha((\!|(\!|K|\!)_{k_{\mathcal{L}}}|\!)_{\top}) = (\!|(\!|K \curvearrowright solve|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}|\!)_{\top} \tag{160}$$
$$\alpha(V : Val) = [\![V]\!]_{\mathcal{T}} \tag{161}$$

Figure 3.15: Definition of α_{Poly}

3.4.4 Proofs

Lemma 14. *If* $\mathcal{T} \models \alpha(V) \xrightarrow{*} \tau$ *then* $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$

Proof. This follows directly from rule 161. □

Lemma 15 (Main Lemma for Preservation)**.** *Let* E *be an expression such that* $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ *and* $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ *for some* τ *and* R*. Then* $\mathcal{T} \models \alpha(R) \xrightarrow{*} \tau$*.*

Proof. The proof proceeds by induction on the number of steps taken to get from $[\![E]\!]_{\mathcal{L}}$ to R.

Base Case Assume no steps were taken. Then $R = [\![E]\!]_{\mathcal{L}}$. By 159 we see that $\alpha(R) = [\![E]\!]_{\mathcal{T}}$. By assumption, this reduces to τ, so we have that $\alpha(R) \xrightarrow{*} \tau$.

Induction case Assume $[\![E]\!]_{\mathcal{L}} \xrightarrow{n} R$ and $\alpha(R) \xrightarrow{*} \tau$. If no steps can be taken from R then the property holds vacuously, so assume an $n+1$ step can be taken to get to a state R'. This step could be any one of the structural or semantic rules of the language. We consider each individually.

Rule 134 $R = [\![E]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![E]\!]_{\mathcal{T}}$	159
$(\!(\!(E \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_0)\!)_{\mathit{nextType}})\!)_{\top}$	147

$R' = [\![(\!(E)\!)_{k_{\mathcal{L}}}]\!]_{\mathcal{L}}$

$\alpha(R')$	By Rule
$(\!(\!(E \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_0)\!)_{\mathit{nextType}})\!)_{\top}$	160

Rule 139 $R = (\!(\!(\lambda X\,.\,E)(V : \mathit{Val}) \curvearrowright K)\!)_{k_{\mathcal{L}}})\!)_{\top}$

$\alpha(R)$	By Rule
$(\!(\!(\lambda X\,.\,E)(V : \mathit{Val}) \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_0)\!)_{\mathit{nextType}}$	160
$(\!(\lambda X\,.\,E \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_0)\!)_{\mathit{nextType}}$	148
$(\!(\tau_0 \to E[\tau_0/X] \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_1)\!)_{\mathit{nextType}}$	152
$(\!(E[\tau_0/X] \curvearrowright \tau_0 \to \Box \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_1)\!)_{\mathit{nextType}}$	150
$(\!(\alpha \curvearrowright \tau_0 \to \Box \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{\mathit{eqns}}\ (\!(\tau_1)\!)_{\mathit{nextType}}$	ind. assm.
$(\!(\tau_0 \to \alpha \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{\mathit{eqns}}\ (\!(\tau_1)\!)_{\mathit{nextType}}$	150
$(\!(\!(\tau_0 \to \alpha)\beta \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{\mathit{eqns}}\ (\!(\tau_1)\!)_{\mathit{nextType}}$	ind. assm.
$(\!(\tau_1 \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\!(\tau_0 \to \alpha)\!)_{\mathit{eqns}} = (\beta \to \tau_1) \cdot \mathcal{E})\ (\!(\tau_2)\!)_{\mathit{nextType}}$	153

We see $\tau_1 = \alpha$ by unification, and that $V \xrightarrow{*} \tau_0$ and $E[\tau_0/X] \xrightarrow{*} \alpha$

$R' = (\!(\!(E[V/X] \curvearrowright K)\!)_{k_{\mathcal{L}}})\!)_{\top}$

$\alpha(R')$	By Rule
$(\!(E[V/X] \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_0)\!)_{\mathit{nextType}}$	160

By a lemma similar to the one we used with Mono (Lemma 11), we know $E[V/X] \xrightarrow{*} \alpha$.

Rule 140 $R = (\!(\!(\mathsf{let}\ X\ \mathsf{be}\ V : \mathit{Val}\ \mathsf{in}\ E \curvearrowright K)\!)_{k_{\mathcal{L}}})\!)_{\top}$

$\alpha(R)$	By Rule
$(\!(\mathsf{let}\ X\ \mathsf{be}\ V : \mathit{Val}\ \mathsf{in}\ E \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_0)\!)_{\mathit{nextType}}$	160
$(\!(E[V/X] \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_0)\!)_{\mathit{nextType}}$	154

$R' = (\!(\!(E[V/X] \curvearrowright K)\!)_{k_{\mathcal{L}}})\!)_{\top}$

$\alpha(R')$	By Rule
$(\!(E[V/X] \curvearrowright K \curvearrowright \mathit{solve})\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{\mathit{eqns}}\ (\!(\tau_0)\!)_{\mathit{nextType}}$	160

Rule 141 $R = (\!|(\!|(C : \mathit{Const})(V : \mathit{Val}) \curvearrowright K|\!)_{k_{\mathcal{L}}}|\!)_{\top}$

$\alpha(R)$	By Rule
$(\!|(C : \mathit{Const})(V : \mathit{Val}) \curvearrowright K \curvearrowright \mathit{solve}|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	160
$(\!|(\mathit{instantiate}(\mathit{TypeOf}(C)))V \curvearrowright K \curvearrowright \mathit{solve}|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	155
$(\!|(\tau_1 \to \tau_2)V \curvearrowright K \curvearrowright \mathit{solve}|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	ind. assm.
$(\!|(\tau_1 \to \tau_2)\tau' \curvearrowright K \curvearrowright \mathit{solve}|\!)_{k_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	ind. assm.
$(\!|\tau_0 \curvearrowright K \curvearrowright \mathit{solve}|\!)_{k_{\mathcal{T}}}\ (\!|(\tau_1 \to \tau_2 = \tau' \to \tau_0) \cdot \mathcal{E}|\!)_{eqns}\ (\!|\tau_1|\!)_{nextType}$	153

$R' = (\!|(\!|\delta(C, V) \curvearrowright K|\!)_{k_{\mathcal{L}}}|\!)_{\top}$

$\alpha(R')$	By Rule
$(\!|\delta(C, V) \curvearrowright K \curvearrowright \mathit{solve}|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	160
$(\!|\tau_2 \curvearrowright K \curvearrowright \mathit{solve}|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	above & δ-typeability

□

3.5 Exp & $\mathscr{W}$

3.5.1 Language Definition

$$
\begin{array}{rcll}
\mathit{Var} & ::= & \text{standard identifiers} & \\
\mathit{Exp} & ::= & \mathit{Var} \mid \ldots \text{ add basic values (Bools, ints, etc.)} & \\
& \mid & \lambda\, \mathit{Var}\,.\, \mathit{Exp} & \\
& \mid & \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
& \mid & \mu\, \mathit{Var}\,.\, \mathit{Exp} & \\
& \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Exp}\ \mathsf{else}\ \mathit{Exp} & [\mathit{strict}(1)] \\
& \mid & \mathsf{let}\ \mathit{Var} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & [\mathsf{let}\ x = e\ \mathsf{in}\ e' = (\lambda x.e')\, e] \\
& \mid & \mathsf{letrec}\ \mathit{Var}\, \mathit{Var} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & [\mathsf{letrec}\ f\ x = e\ \mathsf{in}\ e' = \mathsf{let}\ f = \mu f.(\lambda x.e)\ \mathsf{in}\ e']
\end{array}
$$

(a) Language Syntax

$$
\begin{array}{rcl}
\mathit{Val} & ::= & \lambda\, \mathit{Var}\,.\, \mathit{Exp} \mid \ldots \text{(Bools, ints, etc.)} \\
\mathit{Result} & ::= & \mathit{Val} \\
\mathit{KProper} & ::= & \mu\, \mathit{Var}\,.\, \mathit{Exp} \\
\mathit{Config} & ::= & \mathit{Val} \mid [\![K]\!] \mid (\!| K |\!)_k
\end{array}
$$

(b) Configuration Syntax

$$
\begin{array}{l}
[\![e]\!] = (\!| (\!| e |\!)_k |\!)_\top \\
(\!| (\!| v |\!)_k |\!)_\top = v \\
(\!| \underline{(\lambda x.e)\, v} |\!)_k \\
\quad e[x \leftarrow v] \\
(\!| \underline{\quad \mu\ x.e \quad} |\!)_k \\
\quad e[x \leftarrow \mu\ x.e] \\
\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rightarrow e_1 \\
\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \rightarrow e_2
\end{array}
$$

(c) Language Rules

Figure 3.16: K-style Definition of Exp

$$[\![E]\!]_{\mathcal{L}} = [\![(\!|E|\!)_{k_{\mathcal{L}}}]\!]_{\mathcal{L}} \quad (162)$$

$$[\![(\!|V|\!)_{k_{\mathcal{L}}}]\!]_{\mathcal{L}} = V \quad (163)$$

$$EE' = E \curvearrowright \Box E' \quad (164)$$

$$EE' = E' \curvearrowright E\Box \quad (165)$$

$$\mathsf{let}\ X\ \mathsf{be}\ E\ \mathsf{in}\ E' = E \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \Box\ \mathsf{in}\ E' \quad (166)$$

$$\mathsf{if}\ E\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 = E \curvearrowright \mathsf{if}\ \Box\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 \quad (167)$$

$$K + K' = K \curvearrowright \Box + K' \quad (168)$$

$$K + K' = K' \curvearrowright K + \Box \quad (169)$$

(a) Structural Rules

$$\frac{(\!|I : Int + I' : Int|\!)_{k_{\mathcal{L}}}}{I +_{int} I'} \quad (170)$$

$$\frac{(\!|(\lambda X\,.\,E)(V : Val)|\!)_{k_{\mathcal{L}}}}{E[V/X]} \quad (171)$$

$$\frac{(\!|\mathsf{let}\ X\ \mathsf{be}\ V : Val\ \mathsf{in}\ E|\!)_{k_{\mathcal{L}}}}{E[V/X]} \quad (172)$$

$$\frac{(\!|\mathsf{if\ true\ then}\ E\ \mathsf{else}\ E'|\!)_{k_{\mathcal{L}}}}{E} \quad (173)$$

$$\frac{(\!|\mathsf{if\ false\ then}\ E\ \mathsf{else}\ E'|\!)_{k_{\mathcal{L}}}}{E'} \quad (174)$$

$$\frac{(\!|\quad \mathsf{fix}\ X\,.\,E \quad|\!)_{k_{\mathcal{L}}}}{E[\mathsf{fix}\ X\,.\,E/X]} \quad (175)$$

(b) Semantic Rules

Figure 3.17: Expanded Definition of Exp

3.5.2 Definition of Exp's Type Inferencer ($\mathscr{W}$)

Var	::=	standard identifiers	
Exp	::=	*Var* | ... add basic values (Bools, ints, etc.)	
	|	λ *Var* . *Exp*	
	|	*Exp Exp*	[*strict*]
	|	μ *Var* . *Exp*	
	|	if *Exp* then *Exp* else *Exp*	[*strict*(1)]
	|	let *Var* = *Exp* in *Exp*	[let $x = e$ in $e' = (\lambda x.e')\, e$]
	|	letrec *Var Var* = *Exp* in *Exp*	[letrec $f\ x = e$ in $e' =$ let $f = \mu f.(\lambda x.e)$ in e']

(a) Language Syntax

Result	::=	*Type*	
TEnv	::=	Map[*Name*, *Type*]	
Type	::=	... | *let*(*Type*)	
K	::=	... | *Type* $\rightarrow$ *K*	[*strict*(2)]

(b) Configuration Syntax

Figure 3.18: K-style Definition of $\mathscr{W}$

$[\![e]\!] = (\!|(\!|e|\!)_k\ (\!|\cdot|\!)_{tenv}\ (\!|\cdot|\!)_{eqns}\ (\!|t_0|\!)_{nextType}|\!)_\top$

$(\!|(\!|t|\!)_k\ (\!|\gamma|\!)_{eqns}|\!)_\top = \gamma[t]$

$i \to int,\ true \to bool,\ false \to bool,$ (and similarly for all the other desired basic values)

$(\!|\underset{int}{t_1 + t_2}|\!)_k\ (\!|\underset{t_1 = int,\ t_2 = int}{\cdot}|\!)_{eqns}$ (and similarly for all the other standard operators)

$(\!|\underset{(\gamma[t])[tl \leftarrow tl']}{x}|\!)_k\ (\!|\eta|\!)_{tenv}\ (\!|\gamma|\!)_{eqns}\ (\!|\underset{t_v + |tl|}{t_v}|\!)_{nextType}$ when $\eta[x] = let(t),\ tl = vars(\gamma[t]) - vars(\eta)$ and $tl' = t_v \ldots (t_v + |tl| - 1)$

$(\!|\underset{\eta[x]}{x}|\!)_k\ (\!|\eta|\!)_{tenv}$ when $\eta[x] \neq let(t)$

$(\!|\underset{(t_v \to e) \curvearrowright restore(\eta)}{\lambda x.e}|\!)_k\ (\!|\underset{\eta[x \leftarrow t_v]}{\eta}|\!)_{tenv}\ (\!|\underset{t_v + 1}{t_v}|\!)_{nextType}$

$(\!|\underset{\cdot}{restore(\eta)}|\!)_k (\!|\underset{\eta}{\eta'}|\!)_{tenv}$

$(\!|\underset{t_v}{t_1\ t_2}|\!)_k\ (\!|\underset{t_1 = t_2 \to t_v}{\cdot}|\!)_{eqns}\ (\!|\underset{t_v + 1}{t_v}|\!)_{nextType}$

$(\!|\underset{e \curvearrowright ?_=(t_v) \curvearrowright restore(\eta)}{\mu x.e}|\!)_k\ (\!|\underset{\eta[x \leftarrow t_v]}{\eta}|\!)_{tenv}\ (\!|\underset{t_v + 1}{t_v}|\!)_{nextType}$

$(\!|t \to \underset{\cdot}{?_= t_v}|\!)_k\ (\!|\underset{t_v = t}{\cdot}|\!)_{eqns}$

$(\!|\underset{e \curvearrowright restore(\eta)}{\mathsf{let}\ x = t\ \mathsf{in}\ e}|\!)_k\ (\!|\underset{\eta[x \leftarrow let(t)]}{\eta}|\!)_{tenv}$

$(\!|\underset{t_1}{\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2}|\!)_k\ (\!|\underset{t = bool,\ t_1 = t_2}{\cdot}|\!)_{eqns}$

(c) Typing Rules

Figure 3.18: K-style Definition of $\mathscr{W}$ (cont.)

3.5.3 Equivalence of Milner's and Our $\mathscr{W}$[2]

We would like to make explicit how our rewriting definition is effectively equivalent to Milner's $\mathscr{W}$, up to the additions of some explicit fundamental data types and operators. To do this, it is easiest to look at $\mathscr{J}$, Milner's simplified algorithm, which he proved equivalent to $\mathscr{W}$. Milner's definition of $\mathscr{J}$ is given in Figure 3.20 as a convenience for the reader.

The main questions of equivalence center around recursive calls and their environments, as well as the substitution. We address each concern in turn. $\mathscr{J}$ is a recursive algorithm. It calls itself on subexpressions throughout the computation. We achieve the same effect through the use of strictness attributes and the saving and restoring of environments. Our strictness attributes cause subexpressions to be moved to the front of the computation structure, effectively disabling rules that would apply to the "context," and enabling rules applying to the subexpression itself.

Type environments (also called typed prefixes in Milner's notation) are passed to each call of $\mathscr{J}$. Because we have only one global type environment, it is not immediately obvious that changes to the type environment when evaluating subexpressions cannot affect the remaining computation. In

[2]This section comes from Ellison et al. [2008].

$$[\![E]\!]_{\mathcal{T}} = (\!|(\!|E|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{envs_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}|\!)_{\top} \tag{176}$$

$$\langle\!|(\!|\tau|\!)_{k_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}|\!\rangle_{\top} = \mathcal{E}[\tau] \tag{177}$$

$$EE' = E \curvearrowright \Box E' \tag{178}$$

$$EE' = E' \curvearrowright E\Box \tag{179}$$

$$\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' = K \curvearrowright \mathsf{if}\ \Box\ \mathsf{then}\ S\ \mathsf{else}\ S' \tag{180}$$

$$\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' = S \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ \Box\ \mathsf{else}\ S' \tag{181}$$

$$\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' = S' \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ \Box \tag{182}$$

$$\mathsf{let}\ X\ \mathsf{be}\ K\ \mathsf{in}\ K' = K \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \Box\ \mathsf{in}\ K' \tag{183}$$

$$\tau \to K = K \curvearrowright \tau \to \Box \tag{184}$$

(a) Structural Rules

$$(\!|\tau \curvearrowright \underset{\cdot}{restore(\Gamma)}|\!)_{k_{\mathcal{T}}}\ (\!|\underset{\Gamma}{\_}|\!)_{envs_{\mathcal{T}}} \tag{185}$$

$$I : Int \longrightarrow \textsc{Int} \tag{186}$$

$$\mathsf{true} \longrightarrow \textsc{Bool} \tag{187}$$

$$\mathsf{false} \longrightarrow \textsc{Bool} \tag{188}$$

$$(\!|\underset{(\mathcal{E}[\tau])[tl \leftarrow tl']}{X}|\!)_{k_{\mathcal{T}}}\ (\!|\Gamma|\!)_{envs_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}\ (\!|\underset{\tau_v + |tl|}{\tau_v}|\!)_{nextType} \tag{189}$$

$$\text{if } \Gamma[X] = \mathsf{let}(t), tl = vars(\mathcal{E}[\tau]) - vars(\Gamma) \text{ and } tl' = \tau_v \ldots (\tau_v + |tl| - 1)$$

$$(\!|\underset{(\tau_v \to E) \curvearrowright restore(\Gamma)}{\lambda X\,.\,E}|\!)_{k_{\mathcal{T}}}\ (\!|\underset{\Gamma[X \leftarrow \tau_v]}{\Gamma}|\!)_{envs_{\mathcal{T}}}\ (\!|\underset{next(\tau_v)}{\tau_v}|\!)_{nextType} \tag{190}$$

$$(\!|\underset{\textsc{Int}}{\tau + \tau'}|\!)_{k_{\mathcal{T}}}\ \langle\!|\underset{\tau = \textsc{Int} \cdot \tau' = \textsc{Int}}{\cdot}|\!\rangle_{eqns} \tag{191}$$

$$(\!|\underset{\tau_v}{\tau_1 \tau_2}|\!)_{k_{\mathcal{T}}}\ (\!|\underset{\tau_1 = \tau_2 \to \tau_v}{\cdot}|\!)_{eqns}\ (\!|\underset{next(\tau_v)}{\tau_v}|\!)_{nextType} \tag{192}$$

$$(\!|\underset{E \curvearrowright restore(\Gamma)}{\mathsf{let}\ X\ \mathsf{be}\ \tau\ \mathsf{in}\ E}|\!)_{k_{\mathcal{T}}}\ (\!|\underset{\Gamma[X \leftarrow let(\tau_v)]}{\Gamma}|\!)_{envs_{\mathcal{T}}} \tag{193}$$

$$(\!|\underset{\tau_1}{\mathsf{if}\ \tau\ \mathsf{then}\ \tau_1\ \mathsf{else}\ \tau_2}|\!)_{k_{\mathcal{T}}}\ \langle\!|\underset{\tau = \textsc{Bool} \cdot \tau_1 = \tau_2}{\cdot}|\!\rangle_{eqns} \tag{194}$$

(b) Semantic Rules

Figure 3.19: Expanded Definition of $\mathscr{W}$

$\mathscr{J}(\bar{p}, f) = \tau$

1. If f is x then:
   If λx_σ is active in $\bar{p}$, $\tau := \sigma$.
   If *let* x_σ is active in $\bar{p}$, $\tau = [\beta_i/\alpha_i]E_\sigma$, where α_i are the generic type variables of *let* $x_{E\sigma}$ in $E\bar{p}$, and β_i are new variables.

2. If f is de then:
   $\rho := \mathscr{J}(\bar{p}, d)$; $\sigma := \mathscr{J}(\bar{p}, e)$;
   UNIFY$(\rho, \sigma \to \beta)$; (β new)
   $\tau := \beta$

3. If f is (*if* d *then* e *else* e'), then:
   $\rho := \mathscr{J}(\bar{p}, d)$; UNIFY$(\rho,$ *bool*$)$;
   $\sigma := \mathscr{J}(\bar{p}, e)$;, $\sigma' := \mathscr{J}(\bar{p}, e')$;
   UNIFY(σ, σ'); $\tau := \sigma$

4. If f is $(\lambda x \cdot d)$ then:
   $\rho := \mathscr{J}(\bar{p} \cdot \lambda x_\beta, d)$; ($\beta$ new)
   $\tau := \beta \to \rho$

5. If f is (*fix* $x \cdot d$), then:
   $\rho := \mathscr{J}(\bar{p} \cdot$ *fix* $x_\beta, d)$; (β new)
   UNIFY(β, ρ); $\tau = \beta$

6. If f is (*let* $x = d$ *in* e) then:
   $\rho := \mathscr{J}(\bar{p}, d)$; $\sigma := \mathscr{J}(\bar{p} \cdot$ *let* $x_\rho, e)$;
   $\tau := \sigma$.

UNIFY is a procedure that delivers no result, but has a side effect on a global substitution E. If UNIFY(σ, τ) changes E to E', and if $\mathscr{U}(E\sigma, E\tau) = U$, then $E' = UE$, where $\mathscr{U}$ is a unification generator.

Figure 3.20: Milner's $\mathscr{J}$ Algorithm

Milner's algorithm, this is handled by virtue of passing the environments by value. We ensure this by always placing, at the end of the local computation, a *restore* marker and a copy of the current environment before affecting the environment. Thus, when the local computation is complete, the environment is restored to what it was before starting the subcomputation.

Both definitions keep a single, global substitution, to which restrictions are continually added as side effects. In addition, both only apply the substitution when doing variable lookup. The calls to UNIFY in the application, if/then/else, and fix cases are reflected in our rules by the additional formulas added to the *eqns* configuration item. Indeed, for the rules of Exp (disregarding our extensions with integers), these are the only times we affect the unifier. As an example, let us look at the application de in an environment $\bar{p}$. In $\mathscr{J}$, two recursive calls to $\mathscr{J}$ are made: $\mathscr{J}(\bar{p}, d)$ and $\mathscr{J}(\bar{p}, e)$, whose results are called ρ and σ respectively. Then a restriction to the global unifier is made, equating ρ with $\sigma \to \beta$, with β being a new type variable, and finally β is returned as the type of the expression.

We do a very similar thing. The strictness attributes of the application operator forces evaluation of the arguments d and e first. These eventually transform into $\rho\sigma$. We can then apply a rewrite rule where we end up with a new type β, and add an equation $\rho = \sigma \to \beta$ to the *eqns* configuration item. The evaluations of d and e are guaranteed not to change the environment because we always restore environments upon returning types.

3.5.4 Definition of Abstract Type Inferencer

$$[\![E]\!]_{\widehat{\mathcal{T}}} = [\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|ID|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \tag{195}$$

$$[\![\langle(\!|\tau|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\theta|\!)_{subst}\rangle]\!]_{\widehat{\mathcal{T}}} = (\!|\tau|\!)_{subst} \tag{196}$$

$$EE' = E \curvearrowright \Box E' \tag{197}$$

$$EE' = E' \curvearrowright E\Box \tag{198}$$

$$\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' = K \curvearrowright \mathsf{if}\ \Box\ \mathsf{then}\ S\ \mathsf{else}\ S' \tag{199}$$

$$\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' = S \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ \Box\ \mathsf{else}\ S' \tag{200}$$

$$\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' = S' \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ \Box \tag{201}$$

$$\mathsf{let}\ X\ \mathsf{be}\ K\ \mathsf{in}\ K' = K \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \Box\ \mathsf{in}\ K' \tag{202}$$

$$\tau \to K = K \curvearrowright \tau \to \Box \tag{203}$$

(a) Structural Rules

$\theta : \mathit{TypeVar} \to \mathit{TypeVar}$ which we extend over K, $\mathit{TypeEnv}$, θs, and $\widehat{K}$ in the natural way.

$$\theta \oplus (\tau_1 = \tau_2) = \theta \otimes ((\!|\tau_1|\!)_{subst} = (\!|\tau_2|\!)_{subst})$$

$$\theta \otimes (\tau_1 = \tau_2) = \begin{cases} \theta & \text{if } \tau_1 = \tau_2 \\ \theta[\tau_1 \leftarrow \tau_2] & \text{if } \tau_1 \in \mathit{TypeVar} \\ \theta[\tau_2 \leftarrow \tau_1] & \text{else if } \tau_2 \in \mathit{TypeVar} \\ \theta' \oplus (\tau_1' = \tau_2') \oplus (\tau_1'' = \tau_2'') & \text{if } \tau_1 \neq \tau_2 \text{ and } \tau_1 = \tau_1' \to \tau_1'' \text{ and } \tau_2 = \tau_2' \to \tau_2'' \\ \bot & \text{otherwise} \end{cases}$$

(b) Unification Operator

Figure 3.21: Definition of Abstract Type Inferencer

$$(\!|\tau \curvearrowright \underline{restore(\Gamma)}|\!)_{k_{\widehat{\mathcal{T}}}} \; \underset{\cdot}{(\!|\_|\!)}_{env_{\mathcal{T}}} \tag{204}$$

$$I : Int \longrightarrow \textsc{Int} \tag{205}$$

$$\mathsf{true} \longrightarrow \textsc{Bool} \tag{206}$$

$$\mathsf{false} \longrightarrow \textsc{Bool} \tag{207}$$

$$(\!|\underline{\quad X \quad}|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\overline{\tau[tl \leftarrow tl']}|\!)_{env_{\mathcal{T}}} \tag{208}$$

where $\Gamma[X] = \mathsf{let}(\tau)$, $tl = vars(\tau) - vars(\Gamma)$, and $tl' = |tl|$ fresh type variables

$$(\!|\underline{\quad X \quad}|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\overline{\Gamma[X]}|\!)_{env_{\mathcal{T}}} \text{ where } \Gamma[X] \neq \mathsf{let}(\_) \tag{209}$$

$$(\!|\underline{\quad \lambda X \,.\, E \quad}|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\underline{\quad \Gamma \quad}|\!)_{env_{\mathcal{T}}} \text{ where } \tau_v \text{ is a fresh type variable} \tag{210}$$
$$(\tau_v \to E) \curvearrowright restore(\Gamma) \qquad \Gamma[X \leftarrow \tau_v]$$

$$(\!|\underline{\quad \mathsf{let}\, X \;\mathsf{be}\; \tau \;\mathsf{in}\; E \quad}|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\underline{\quad \Gamma \quad}|\!)_{env_{\mathcal{T}}} \text{ where } \tau_v \text{ is a fresh type variable} \tag{211}$$
$$E \curvearrowright restore(\Gamma) \qquad \Gamma[X \leftarrow let(\tau_v)]$$

$$(\!|\underline{\quad \mathsf{fix}\; X \,.\, E \quad}|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\underline{\quad \Gamma \quad}|\!)_{env_{\mathcal{T}}} \text{ where } \tau_v \text{ is a fresh type variable} \tag{212}$$
$$E \curvearrowright ?_{=}(\tau_v) \curvearrowright restore(\Gamma) \qquad \Gamma[X \leftarrow \tau_v]$$

$$(\!|\tau \curvearrowright \underset{\cdot}{\underline{?_{=}(\tau_v)}}|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\underline{\quad \theta \quad}|\!)_{subst} \tag{213}$$
$$\theta \oplus (\tau = \tau_v)$$

$$(\!|\underline{\tau + \tau'}|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\underline{\quad \theta \quad}|\!)_{subst} \tag{214}$$
$$\textsc{Int} \qquad \theta \oplus (\tau = \textsc{Int}) \oplus (\tau' = \textsc{Int})$$

$$(\!|\underline{\tau_1 \tau_2}|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\underline{\quad \theta \quad}|\!)_{subst} \text{ where } \tau_v \text{ is a fresh type variable} \tag{215}$$
$$\tau_v \qquad \theta \oplus (\tau_1 = \tau_2 \to \tau_v)$$

$$(\!|\underline{\mathsf{if}\; \tau \;\mathsf{then}\; \tau_1 \;\mathsf{else}\; \tau_2}|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\underline{\theta}|\!)_{subst} \text{ where } \theta' = \theta \oplus (\tau_1 = \tau_2) \oplus (\tau = \textsc{Bool}) \tag{216}$$
$$\tau_1 \qquad \theta'$$

(c) Structural Rules

Figure 3.21: Definition of Abstract Type Inferencer (cont.)

We define the reduction relation $\xrightarrow{\widehat{}}$ as $\xrightarrow{\widehat{}} \;=\; \longrightarrow; \theta$. We further identify configurations $[\![(\!|K_1|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\Gamma_1|\!)_{env_{\mathcal{T}}} \; (\!|\theta_1|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$ and $[\![(\!|K_2|\!)_{k_{\widehat{\mathcal{T}}}} \; (\!|\Gamma_2|\!)_{env_{\mathcal{T}}} \; (\!|\theta_2|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$ where $\exists$ a bijection $\iota : \mathit{TypeVar} \to \mathit{TypeVar}$, extended in the usual way, such that $\iota(K_1) = K_2$, $\iota(\Gamma_1) = \Gamma_2$, and $\iota(\theta_1) = \theta_2$.

Lemma 16. *Any reachable configuration in the language domain can be transformed using structural rules into a unique expression.*

Proof. This follows from two key points. One, you cannot use the structural rules to transform an expression into any other expression, and two, each structural rule can be applied backwards even after semantic rules have applied. □

3.5.5 Abstraction Function Definition

$$\alpha([\![E]\!]_{\mathcal{L}}) = [\![E]\!]_{\mathcal{T}} \tag{217}$$

Figure 3.22: Definition of α_{Exp}

By Lemma 16, we know this definition of α is well-defined for all reachable configurations, and homomorphic with respect to structural rules.

3.5.6 Proofs

The proofs in this section that describe properties of the abstract type inferencer are preliminary. They represent mostly a collection of lemmas we believe to be true, but they have not been formalized yet.

Definition 1 (Generalizes relation)**.** A type τ is said to *generalize* a type τ' (written $\tau \succ \tau'$) if $\exists\ \theta\ \mid\ \theta(\tau) = \tau'$ where θ is a substitution over type variables.

Definition 2 (Type Equivalance)**.** A type τ is said to be *type equivalent* to τ' if $\tau \succ \tau'$ and $\tau' \succ \tau$.

Lemma 17.

$$[\![(\!|K_1|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{}} [\![(\!|K_2|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma'|\!)_{env_{\mathcal{T}}}\ (\!|\theta'|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

iff

$$[\![(\!|K_1 \curvearrowright K|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{}} [\![(\!|K_2 \curvearrowright \theta'(K)|\!)_{k_{\widehat{\mathcal{T}}}})\ (\!|\Gamma'|\!)_{env_{\mathcal{T}}}\ (\!|\theta'|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

Lemma 18. $\theta \succ \theta \oplus \mathcal{E}$

Lemma 19. *If*

$$[\![(\!|K|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{}} [\![(\!|K'|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma'|\!)_{env_{\mathcal{T}}}\ (\!|\theta'|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

then $\theta \succ \theta'$.

Lemma 20. *If*

$$[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma'|\!)_{env_{\mathcal{T}}}\ (\!|\theta'|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

then $\Gamma' = \theta'(\Gamma)$.

Lemma 21. *If*

$$[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_1|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma'|\!)_{env_{\mathcal{T}}}\ (\!|\theta'|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

then if

$$[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma[X \leftarrow \tau]|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_2|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma''|\!)_{env_{\mathcal{T}}}\ (\!|\theta''|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

for some fresh type variable τ*, we have that* $\tau_2 \succ \tau_1$.

Lemma 22. *If*

$$[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma[X \leftarrow \tau]|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_1|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma'|\!)_{env_{\mathcal{T}}}\ (\!|\theta'|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

then if

$$[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma[X \leftarrow \tau']|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_2|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma''|\!)_{env_{\mathcal{T}}}\ (\!|\theta''|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

for types $\tau' \succ \tau$*, we have that* $\tau_2 \succ \tau_1$.

Lemma 23. *If*

$$[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_1|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\theta'(\Gamma)|\!)_{env_{\mathcal{T}}}\ (\!|\theta'|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

and E contains no X, then

$$[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma[X \leftarrow \tau]|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_1|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\theta'(\Gamma)[X \leftarrow \tau]|\!)_{env_{\mathcal{T}}}\ (\!|\theta'|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

for a fresh τ.

Lemma 24. *If*

$$[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_1|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma_1|\!)_{env_{\mathcal{T}}}\ (\!|\theta_1|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

and τ is a fresh type variable, then for any X,

$$[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma[X \leftarrow \tau]|\!)_{env_{\mathcal{T}}}\ (\!|\theta|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_2|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma_2|\!)_{env_{\mathcal{T}}}\ (\!|\theta_2|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$$

and $\tau_2 \succ \tau_1$ and $\theta_2 \succ \theta_1$.

Lemma 25 (Env $\Rightarrow$ Rep)**.** *If*

- $[\![(\!|V|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta_0|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_V|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma_1|\!)_{env_{\mathcal{T}}}\ (\!|\theta_1|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$
- $[\![(\!|E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma_1[X \leftarrow \tau_V]|\!)_{env_{\mathcal{T}}}\ (\!|\theta_1|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma_2|\!)_{env_{\mathcal{T}}}\ (\!|\theta_2|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$

then $[\![(\!|E[X \leftarrow V]|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta_0|\!)_{subst}]\!]_{\widehat{\mathcal{T}}} \xrightarrow{\hat{*}} [\![(\!|\tau_R|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma_3|\!)_{env_{\mathcal{T}}}\ (\!|\theta_3|\!)_{subst}]\!]_{\widehat{\mathcal{T}}}$ where $\tau_R \succ \tau_E$ and $\theta_3 \succ \theta_2$.

Proof. We will do an induction on the size (number of operators) of E. (Incidentally, $\Gamma_1 = \theta_1(\Gamma)$, $\Gamma_2 = \theta_2(\Gamma_1[X \leftarrow \tau_V])$, and $\Gamma_3 = \theta_3(\Gamma)$.

Base Case: Consider an expression E of size 0. These include only variable names and constants.

Inductive Case: Consider an expression E of size $n+1$. It could be any of the expressions of the language, and we consider each in turn. Assume the above property holds for all expressions up to size n.

Consider the case when E is a lambda expression. We assume

$$(\!|V|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta_0|\!)_{subst} \xrightarrow{\hat{*}} (\!|\tau_V|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma_1|\!)_{env_{\mathcal{T}}}\ (\!|\theta_1|\!)_{subst} \tag{218}$$

and

$$(\!|\lambda Y\,.\,E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma_1[X \leftarrow \tau_V]|\!)_{env_{\mathcal{T}}}\ (\!|\theta_1|\!)_{subst} \xrightarrow{\hat{*}} (\!|E \curvearrowright K|\!)_{k_{\widehat{\mathcal{T}}}} (\!|\Gamma_1[X \leftarrow \tau_V][Y \leftarrow \tau]|\!)_{env_{\mathcal{T}}} (\!|\theta_1|\!)_{subst} \tag{219}$$

$$\xrightarrow{\hat{*}} (\!|\tau_E \curvearrowright \theta_2(K)|\!)_{k_{\widehat{\mathcal{T}}}})\ (\!|\Gamma'|\!)_{env_{\mathcal{T}}}\ (\!|\theta_2|\!)_{subst} \tag{220}$$

$$\xrightarrow{\hat{*}} (\!|\theta_2(\tau) \to \tau_E|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma''|\!)_{env_{\mathcal{T}}}\ (\!|\theta_2|\!)_{subst} \tag{221}$$

$$\tag{222}$$

for $K = (\tau \to \square) \curvearrowright \mathit{restore}(\Gamma_1[X \leftarrow \tau_V])$. We want to show that

$$(\!|(\lambda Y\,.\,E)[X \leftarrow V]|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma|\!)_{env_{\mathcal{T}}}\ (\!|\theta_0|\!)_{subst} \xrightarrow{\hat{*}} (\!|\tau_R|\!)_{k_{\widehat{\mathcal{T}}}}\ (\!|\Gamma_3|\!)_{env_{\mathcal{T}}}\ (\!|\theta_3|\!)_{subst} \tag{223}$$

with $\tau_R \succ \theta_2(\tau) \to \tau_E$ and $\theta_3 \succ \theta_2$.

We first notice that there can be no Y in V, so by Lemma 23 and assumption 218, we know that

$$(\!|V|\!)_{k_{\hat{\mathcal{T}}}}\ (\!|\Gamma[\,Y \leftarrow \tau]|\!)_{env_{\mathcal{T}}}\ (\!|\theta_0|\!)_{subst} \xrightarrow{\hat{*}} (\!|\tau_V|\!)_{k_{\hat{\mathcal{T}}}}\ (\!|\Gamma_1[\,Y \leftarrow \tau]|\!)_{env_{\mathcal{T}}}\ (\!|\theta_1|\!)_{subst} \tag{224}$$

Furthermore, by Lemma 24,

$$(\!|E|\!)_{k_{\hat{\mathcal{T}}}}\ (\!|\Gamma_1[\,Y \leftarrow \tau][X \leftarrow \tau_V]|\!)_{env_{\mathcal{T}}}\ (\!|\theta_1|\!)_{subst} \xrightarrow{\hat{*}} (\!|\tau'_E|\!)_{k_{\hat{\mathcal{T}}}}\ (\!|\Gamma'_2|\!)_{env_{\mathcal{T}}}\ (\!|\theta\,'_2|\!)_{subst} \tag{225}$$

for some $\tau'_E \succ \tau_E$ and $\theta\,'_2 \succ \theta_2$. Now we can apply the inductive hypothesis to conclude

$$(\!|E[X \leftarrow V]|\!)_{k_{\hat{\mathcal{T}}}}\ (\!|\Gamma[\,Y \leftarrow \tau]|\!)_{env_{\mathcal{T}}}\ (\!|\theta_0|\!)_{subst} \xrightarrow{\hat{*}} (\!|\tau_S|\!)_{k_{\hat{\mathcal{T}}}}\ (\!|\Gamma_3|\!)_{env_{\mathcal{T}}}\ (\!|\theta_3|\!)_{subst} \tag{226}$$

with $\tau_S \succ \tau'_E \succ \tau_E$ and $\theta_3 \succ \theta\,'_2 \succ \theta_2$.

So now we know that

$$(\!|(\lambda\, Y\,.\, E)[X \leftarrow V]|\!)_{k_{\hat{\mathcal{T}}}}(\!|\Gamma|\!)_{env_{\mathcal{T}}}(\!|\theta_0|\!)_{subst} \xrightarrow{\hat{*}} (\!|E[X \leftarrow V] \curvearrowright K|\!)_{k_{\hat{\mathcal{T}}}}(\!|\Gamma[\,Y \leftarrow \tau]|\!)_{env_{\mathcal{T}}}(\!|\theta_0|\!)_{subst} \tag{227}$$

$$\xrightarrow{\hat{*}} (\!|\theta_3(\tau) \to \tau_S|\!)_{k_{\hat{\mathcal{T}}}}(\!|\Gamma_3|\!)_{env_{\mathcal{T}}}(\!|\theta_3|\!)_{subst} \tag{228}$$

All that remains to be shown is that $\theta_3(\tau) \to \tau_S \succ \theta_2(\tau) \to \tau_E$. We know:

1. $\exists\theta.\forall\tau_v.\theta(\theta_3(\tau_v)) = \theta_2(\tau_v)$
2. $\exists\theta'.\theta'(\tau_S) = \tau_E$
3. $\theta_2(\tau_E) = \tau_E$
4. $\theta_3(\tau_S) = \tau_S$
5. τ is a base type

This is true if $\exists\theta''.\theta''(\theta_3(\tau) \to \tau_S) = \theta_2(\tau) \to \tau_E$, or equivalently we can find a θ'' st

1. $\theta''(\theta_3(\tau)) = \theta_2(\tau)$
2. $\theta''(\tau_S) = \tau_E$

We have not been able to show this yet.

□

Lemma 26 (Top of Stack Lemma)**.** *If* $(\!|S \curvearrowright K|\!)_{k_{\mathcal{T}}} \xrightarrow{*} (\!|\tau \curvearrowright K|\!)_{k_{\mathcal{T}}}$*, then* $(\!|S \curvearrowright K'|\!)_{k_{\mathcal{T}}} \xrightarrow{*} (\!|\tau \curvearrowright K|\!)_{k_{\mathcal{T}}}$ *for any* K'*.*

Lemma 27. *If* $\mathcal{T} \models \alpha(V) \xrightarrow{*} \tau$ *then* $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$

Proof. This follows directly from the $\mathscr{W}$ rewrite rules for values. □

Lemma 28 (Main Lemma for Preservation)**.** *If* $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ *and* $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ *for some* τ *and* R*, then* $\mathcal{T} \models \alpha(R) \xrightarrow{*} \tau'$ *for some* τ' *unifiable with* τ*.*

Proof. The proof proceeds by induction on the number of steps taken to get from $[\![E]\!]_{\mathcal{L}}$ to R.

Base Case Assume no steps were taken. Then $R = [\![E]\!]_{\mathcal{L}}$. By the definition of α, we see that $\alpha(R) = [\![E]\!]_{\mathcal{T}}$. By assumption, this reduces to τ, so we have that $\alpha(R) \xrightarrow{*} \tau$.

Induction Case Assume $[\![E]\!]_{\mathcal{L}} \xrightarrow{n} R$ and $\alpha(R) \xrightarrow{*} \tau$. If no steps can be taken from R then the property holds vacuously, so assume an $n+1$ step can be taken to get to a state R'. This step could be any one of the structural or semantic rules of the language. We consider each individually:

Rule 162 through 169 These all follow from Lemma 16.

Rule 170 $R = (\!(\!(I : Int + I' : Int \curvearrowright K)\!)_{k_{\mathcal{L}}}\!)_{\top}$ Now we work with $\alpha(R)$:

$$\alpha(R) = \alpha((\!(\!(I + I' \curvearrowright K)\!)_{k_{\mathcal{L}}}\!)_{\top})$$

which reduces to:

$$(\!(\!(I + I' \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}\!)_{\top}$$

by the definition of α. This then reduces to:

$$(\!(\!(\textsc{Int} + \textsc{Int} \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}\!)_{\top}$$

because we reduce integers to Int. This then reduces to:

$$(\!(\!(\textsc{Int} \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\textsc{Int} = \textsc{Int}, \textsc{Int} = \textsc{Int})\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}\!)_{\top}$$

by applying the reduction rule for addition. Finally, we can reduce this to:

$$(\!(\!(\textsc{Int} \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}\!)_{\top}$$

by applying one of the rules of unification twice. Now we work with R'. We start with:

$$\alpha(R') = \alpha((\!(\!(I +_{int} I' \curvearrowright K)\!)_{k_{\mathcal{L}}}\!)_{\top})$$

which reduces to:

$$(\!(\!(I +_{int} I' \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}\!)_{\top}$$

by the definition of α. This immediately reduces to:

$$(\!(\!(\textsc{Int} \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}\!)_{\top}$$

because we reduce integers to Int. So, we now have that $\alpha(R)$ and $\alpha(R')$ both reduce to the same configuration. We know by inductive assumption that $\alpha(R) \xrightarrow{*} \tau$. Since $\alpha(R)$ and $\alpha(R')$ both reduce to the same configuration, $\alpha(R) \xrightarrow{*} \tau$ also. This completes the case.

Rule 171 $R = (\!|(\!|(\lambda X\,.\,E)(V : \mathit{Val}) \curvearrowright K|\!)_{k_{\mathcal{L}}}|\!)_{\top}$

$\alpha(R)$	By Rule
$(\!|(\lambda X\,.\,E)(V : \mathit{Val}) \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	217
$(\!|\lambda X\,.\,E \curvearrowright \Box V \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	197
$(\!|(\tau_0 \to E) \curvearrowright \mathit{restore}(\cdot) \curvearrowright \Box V \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|[X, \tau_0]|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_1|\!)_{nextType}$	210
$(\!|E \curvearrowright (\tau_0 \to \Box) \curvearrowright \mathit{restore}(\cdot) \curvearrowright \Box V \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|[X, \tau_0]|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_1|\!)_{nextType}$	203
$(\!|\tau_1 \curvearrowright (\tau_0 \to \Box) \curvearrowright \mathit{restore}(\cdot) \curvearrowright \Box V \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|[X, \tau_0]|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}\ (\!|\tau_v|\!)_{nextType}$	ind. assm.
$(\!|(\tau_0 \to \tau_1) \curvearrowright \mathit{restore}(\cdot) \curvearrowright \Box V \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|[X, \tau_0]|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}\ (\!|\tau_v|\!)_{nextType}$	203
$(\!|(\tau_0 \to \tau_1) \curvearrowright \Box V \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}\ (\!|\tau_v|\!)_{nextType}$	204
$(\!|(\tau_0 \to \tau_1)V \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}\ (\!|\tau_v|\!)_{nextType}$	197
$(\!|V \curvearrowright (\tau_0 \to \tau_1)\Box \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}\ (\!|\tau_v|\!)_{nextType}$	198
$(\!|\tau_2 \curvearrowright (\tau_0 \to \tau_1)\Box \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}'|\!)_{eqns}\ (\!|\tau_v{}'|\!)_{nextType}$	ind. assm.
$(\!|(\tau_0 \to \tau_1)\tau_2 \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}'|\!)_{eqns}\ (\!|\tau_v{}'|\!)_{nextType}$	198
$(\!|\tau_v{}' \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}' \cdot (\tau_0 \to \tau_1 = \tau_2 \to \tau_v{}')|\!)_{eqns}\ (\!|\tau_v{}'|\!)_{nextType}$	215

In the above we see that

- $(\!|E \curvearrowright K_1|\!)_{k_{\mathcal{T}}}\ (\!|[X, \tau_0]|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_1|\!)_{nextType} \xrightarrow{*}$ $(\!|\tau_1 \curvearrowright K_1|\!)_{k_{\mathcal{T}}}\ (\!|[X, \tau_0]|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}\ (\!|\tau_v|\!)_{nextType}$
- $(\!|V \curvearrowright K_2|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}|\!)_{eqns}\ (\!|\tau_v|\!)_{nextType} \xrightarrow{*}$ $(\!|\tau_2 \curvearrowright K_2|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\mathcal{E}'|\!)_{eqns}\ (\!|\tau_v{}'|\!)_{nextType}$.
- $\tau_0 = \tau_2$
- $\tau_1 = \tau_v{}'$

$R' = (\!|(\!|E[V/X] \curvearrowright K|\!)_{k_{\mathcal{L}}}|\!)_{\top}$

$\alpha(R')$	By Rule
$(\!|E[V/X] \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	217
$(\!|\tau_v{}' \curvearrowright K|\!)_{k_{\mathcal{T}}}\ (\!|\cdot|\!)_{env_{\mathcal{T}}}\ (\!|\cdot|\!)_{eqns}\ (\!|\tau_0|\!)_{nextType}$	Lemma 25

Rule 172 $R = (\!(\!(\mathsf{let}\ X\ \mathsf{be}\ V : \mathit{Val}\ \mathsf{in}\ E \curvearrowright K)\!)_{k_{\mathcal{L}}})_{\top}$

$\alpha(R)$	By Rule
$(\!(\mathsf{let}\ X\ \mathsf{be}\ V\ \mathsf{in}\ E \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}$	217
$(\!(V \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \square\ \mathsf{in}\ E \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}$	202
$(\!(\tau_1 \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \square\ \mathsf{in}\ E \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_v)\!)_{nextType}$	ind. assm.
$(\!(\mathsf{let}\ X\ \mathsf{be}\ \tau_1\ \mathsf{in}\ E \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{eqns}\ (\!(\tau_v)\!)_{nextType}$	202
$(\!(E \curvearrowright \mathit{restore}(\cdot) \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!([X, \mathit{let}(\tau_1)])\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{eqns}\ (\!(\tau_v)\!)_{nextType}$	211
$(\!(\tau_2 \curvearrowright \mathit{restore}(\cdot) \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!([X, \mathit{let}(\tau_1)])\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E}')\!)_{eqns}\ (\!(\tau_v{}')\!)_{nextType}$	ind. assm.
$(\!(\tau_2 \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E}')\!)_{eqns}\ (\!(\tau_v{}')\!)_{nextType}$	204

In the above we see that

- $(\!(E \curvearrowright K_1)\!)_{k_{\mathcal{T}}}\ (\!([X, \mathit{let}(\tau_1)])\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{eqns}\ (\!(\tau_v)\!)_{nextType} \xrightarrow{*}$ $(\!(\tau_2 \curvearrowright K_1)\!)_{k_{\mathcal{T}}}\ (\!([X, \mathit{let}(\tau_1)])\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E}')\!)_{eqns}\ (\!(\tau_v{}')\!)_{nextType}$
- $(\!(V \curvearrowright K_2)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{eqns}\ (\!(\tau_0)\!)_{nextType} \xrightarrow{*}$ $(\!(\tau_1 \curvearrowright K_2)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E}')\!)_{eqns}\ (\!(\tau_v)\!)_{nextType}$.

$R' = (\!(\!(E[V/X] \curvearrowright K)\!)_{k_{\mathcal{L}}})_{\top}$

$\alpha(R')$	By Rule
$(\!(E[V/X] \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}$	217
$(\!(\tau_v{}' \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}$	Lemma 25

Rule 173 $R = (\!(\!(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ E\ \mathsf{else}\ E' \curvearrowright K)\!)_{k_{\mathcal{L}}})_{\top}$

$\alpha(R)$	By Rule
$(\!(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ E\ \mathsf{else}\ E' \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}$	217
$(\!(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ E\ \mathsf{else}\ E' \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}$	206
$(\!(E \curvearrowright \mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \square\ \mathsf{else}\ E' \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}$	200
$(\!(\tau \curvearrowright \mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \square\ \mathsf{else}\ E' \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{eqns}\ (\!(\tau_v)\!)_{nextType}$	ind. assm.
$(\!(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \tau\ \mathsf{else}\ E' \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{eqns}\ (\!(\tau_v)\!)_{nextType}$	200
$(\!(E' \curvearrowright \mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \tau\ \mathsf{else}\ \square \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{eqns}\ (\!(\tau_v)\!)_{nextType}$	201
$(\!(\tau' \curvearrowright \mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \tau\ \mathsf{else}\ \square \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E}')\!)_{eqns}\ (\!(\tau_v{}')\!)_{nextType}$	ind. assm.
$(\!(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \tau\ \mathsf{else}\ \tau' \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E}')\!)_{eqns}\ (\!(\tau_v{}')\!)_{nextType}$	201
$(\!(\tau \curvearrowright K)\!)_{k_{\mathcal{L}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E}' \cdot \textsc{Bool} = \textsc{Bool} \cdot \tau = \tau')\!)_{eqns}\ (\!(\tau_v{}')\!)_{nextType}$	216

$R' = (\!(\!(E \curvearrowright K)\!)_{k_{\mathcal{L}}})_{\top}$

$\alpha(R')$	By Rule
$(\!(E \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\cdot)\!)_{eqns}\ (\!(\tau_0)\!)_{nextType}$	217
$(\!(\tau \curvearrowright K)\!)_{k_{\mathcal{T}}}\ (\!(\cdot)\!)_{env_{\mathcal{T}}}\ (\!(\mathcal{E})\!)_{eqns}\ (\!(\tau_v)\!)_{nextType}$	Lemma 26

Rule 174 This proceeds like rule 173.

□

Theorem 4 (Preservation)**.** *If* $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ *and* $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} V$ *for some type* τ *and value* V*, then* $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau'$ *unifiable with* τ*.*

Proof. This follows directly from Lemmas 27 and 28. □

Chapter 4

Conclusions

4.1 Statement of Results

The aim of this work was to demonstrate the feasibility of proving type preservation using the K-method. We were able to sketch formal arguments for preservation of typing in five different languages and type systems. This result is promising, and suggests work in this area should continue. However, much more complicated type systems need to be investigated before we can say whether the technique composes or even scales.

4.2 Problems Left Unsolved

This work does not address languages that have references, continuations, halt, exceptions, records, reflection, and many other common (and typically problematic) language features. It is important to note, however, that all of these language features have been investigated on their own, but not in relation to type systems. Similarly, many type system features have also been neglected—for example, no effort has been made in type systems with subtyping. We plan on starting work soon for some of these features (records and subtyping) for our submission to PoplMark [Aydemir et al., 2005].

This work also does not address the issue of modularity. The modularity of K at the language-definition level has been established by [Roşu, 2006, 2008]. It is highly desirable that type system definitions be modular. If one adds subtyping, one should not need to do anything but add those particular rules. Additionally, it would be ideal to have the proofs of language level theorems to be as modular as the language semantics. To study this, one would need to take an arbitrary language and type system, for which preservation has been proved, and add either new language or type system features. Then a careful analysis can be made as to which parts of the original definitions and proofs can be kept.

This proof is also not as formal as a proof about programming languages should be. We should be able to formalize this work in a proof assistant, and have the computer aid us in proving these and similar properties. Additionally, a library of commonly used theorems should accompany the K-prelude of commonly used programming language features.

References

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *TPHOLs '05*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005. ISBN 3-540-28372-2. URL `http://dx.doi.org/10.1007/11541868_4`.

R. Banach. Simple type inference for term graph rewriting systems. In *CTRS '92*, volume 656 of *LNCS*, pages 51–66, 1992. ISBN 3-540-56393-8.

H. Barendregt. Introduction to generalized type systems. *J. Functional Programming*, 1(2):125–154, 1991.

H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In *PARLE (2)*, volume 259 of *LNCS*, pages 141–158. Springer, 1987. ISBN 3-540-17945-3.

G. Berry and G. Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4. URL `http://doi.acm.org/10.1145/96709.96717`.

M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. *Stratego/XT Tutorial, Examples, and Reference Manual*. Department of Information and Computing Sciences, Universiteit Utrecht, August 2005. (Draft).

F. Chen, M. Hills, and G. Roşu. A rewrite logic approach to semantic definition, design and analysis of object-oriented languages. Technical Report UIUCDCS-R-2006-2702, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002. ISSN 0304-3975. URL `http://dx.doi.org/10.1016/S0304-3975(01)00359-0`.

H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

C. Ellison, T. F. Şerbănuţă, and G. Roşu. A rewriting logic approach to type inference: Technical report. Technical Report UIUCDCS-R-2008-2934, Computer Science Department, University of Illinois at Urbana-Champaign, 2008.

M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *J. TCS*, 103(2):235–271, 1992. URL `http://citeseer.ist.psu.edu/felleisen92revised.html`.

S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: Indexed types now! In *PEPM '07*, pages 112–121. ACM, 2007. ISBN 978-1-59593-620-2. URL `http://doi.acm.org/10.1145/1244381.1244400`.

M. Hills and G. Roşu. KOOL: A K-based object-oriented language. Technical Report UIUCDCS-R-2006-2779, Computer Science Department, University of Illinois at Urbana-Champaign, 2006.

M. Hills and G. Roşu. A rewriting approach to the design and evolution of object-oriented languages. Technical Report Bericht-Nr. 2007-7, Fakultt IV - Elektrotechnik und Informatik, Technische Universitt Berlin, 2007a. URL http://fsl.cs.uiuc.edu/pubs/hills-rosu-2007-ecoopds.pdf.

M. Hills and G. Roşu. KOOL: An application of rewriting logic to language prototyping and analysis. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA'07)*, volume 4533 of *LNCS*, pages 246–256. Springer, 2007b. URL http://fsl.cs.uiuc.edu/pubs/hills-rosu-2007-rta.pdf.

M. Hills, T. F. Şerbănuţă, and G. Roşu. A rewrite framework for language definitions and for generation of efficient interpreters. In *6th International Workshop on Rewriting Logic and its Applications (WRLA'06)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 215–231. Elsevier Science, July 2007. URL http://fsl.cs.uiuc.edu/pubs/hills-serbanuta-rosu-2006-wrla.pdf. Also appeared as Technical Report UIUCDCS-R-2005-2667, December 2005.

R. Hosabettu, M. K. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In *CAV '98*, volume 1427 of *LNCS*, pages 122–134. Springer, 1998. ISBN 3-540-64608-6.

Y. Huencke and O. de Moor. Aiding dependent type checking with rewrite rules, 2001. URL http://citeseer.ist.psu.edu/huencke01aiding.html.

F. Kamareddine. EPSRC project GR/L36963, 1997–2000. URL http://www.macs.hw.ac.uk/~fairouz/projects/TYREprog.html.

F. Kamareddine and J. W. Klop, editors. *Special issue on Type Theory and Term Rewriting: A collection of papers*, volume 10(3), 2000. Oxford University Press.

F. Kamareddine and J. Wells. On type theory and term rewriting for expressive and efficient programming languages, EPSRC project GR/L36963, one page summary, 1997–2000. URL http://www.macs.hw.ac.uk/~fairouz/projects/tyre-one.ps.

A. Kanade, A. Sanyal, and U. P. Khedker. A PVS based framework for validating compiler optimizations. In *SEFM '06*, pages 108–117. IEEE Computer Society, 2006. ISBN 0-7695-2678-0. URL http://doi.ieeecomputersociety.org/10.1109/SEFM.2006.4.

A. Kanade, A. Sanyal, and U. P. Khedker. Structuring optimizing transformations and proving them sound. In *COCV '06*, volume 176(3) of *ENTCS*, pages 79–95. Elsevier, 2007. URL http://dx.doi.org/10.1016/j.entcs.2006.06.018.

G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *TOPLAS*, 28(4):619–695, 2006. URL http://doi.acm.org/10.1145/1146809.1146811.

J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–117. Oxford University Press, 1992.

G. Kuan, D. MacQueen, and R. B. Findler. A rewriting semantics for type inference. In *ESOP '07*, volume 4421 of *LNCS*, pages 426–440. Springer, 2007. ISBN 978-3-540-71314-2. URL http://dx.doi.org/10.1007/978-3-540-71316-6_29.

D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *POPL '07*, pages 173–184. ACM, 2007. ISBN 1-59593-575-4. URL http://doi.acm.org/10.1145/1190216.1190245.

M. Y. Levin and B. C. Pierce. TinkerType: a language for playing with formal systems. *J. Functional Programing*, 13(2):295–316, 2003. URL http://dx.doi.org/10.1017/S0956796802004550.

A. Mametjanov. Types and program transformations. In *OOPSLA '07 Companion*, pages 937–938. ACM, 2007. ISBN 978-1-59593-865-7. URL http://doi.acm.org/10.1145/1297846.1297954.

J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *RTA '04*, volume 3091 of *LNCS*, pages 301–311. Springer, 2004. ISBN 3-540-22153-0. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3091&spage=301.

P. Meredith, M. Hills, and G. Roşu. An executable rewriting logic semantics of K-Scheme. In D. Dube, editor, *Proceedings of the 2007 Workshop on Scheme and Functional Programming*. Laval University, 2007.

J. Meseguer. Conditioned rewriting logic as a unified model of concurrency. *J. TCS*, 96(1):73–155, 1992.

J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In *IJCAR '04*, volume 3097 of *LNCS*, pages 1–44. Springer, 2004. ISBN 3-540-22345-2. URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3097&spage=1.

J. Meseguer and G. Roşu. The rewriting logic semantics project. *J. TCS*, 373(3):213–237, 2007. ISSN 0304-3975. URL http://dx.doi.org/10.1016/j.tcs.2006.12.018. Also appeared in *SOS '05*, volume 156(1) of *ENTCS*, pages 27–56, 2006.

R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17 (3):348–375, 1978.

G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.

D. Plump. Term graph rewriting, 1998. URL http://citeseer.ist.psu.edu/plump98term.html.

G. Roşu. CS322, Fall 2003: Programming language design: Lecture notes. Technical Report UIUCDCS-R-2003-2897, University of Illinois at Urbana-Champaign, Department of Computer Science, December 2003. URL http://fsl.cs.uiuc.edu/pubs/UIUCDCS-R-2003-2897.pdf. Lecture notes of a course taught at UIUC.

G. Roşu. K: A rewrite-based framework for modular language design, semantics, analysis and implementation. Technical Report UIUCDCS-R-2005-2672, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005. URL http://fsl.cs.uiuc.edu/pubs/rosu-2005-tr.pdf.

G. Roşu. K: A rewrite-based framework for modular language design, semantics, analysis and implementation. Technical Report UIUCDCS-R-2006-2802, Computer Science Department, University of Illinois at Urbana-Champaign, 2006. URL http://fsl.cs.uiuc.edu/pubs/rosu-2006-tr-c.pdf.

G. Roşu. K: A rewriting-based framework for computations: Preliminary version. Technical Report Department of Computer Science UIUCDCS-R-2007-2926 and College of Engineering UILU-ENG-2007-1827, University of Illinois at Urbana-Champaign, 2007. URL http://fsl.cs.uiuc.edu/pubs/rosu-2007-tr-c.pdf.

G. Roşu. K: A rewriting-based framework for computations: An informal guide. Unpublished, 2008.

T. F. Şerbănuţă and G. Roşu. Computationally equivalent elimination of conditions - extended abstract. In *Proceedings of Rewriting Techniques and Applications (RTA'06)*, volume 4098 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2006. URL http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2006-rta.pdf. Also appeared as Technical Report UIUCDCS-R-2006-2693, February 2006.

M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic: Specifying typed higher-order languages in a first-order logical framework. In *Essays in Memory of Ole-Johan Dahl*, volume 2635 of *LNCS*, pages 334–375. Springer, 2004. ISBN 3-540-21366-X. URL `http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2635&spage=334`.

D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In *Formal Syntax and Semantics of Java*, pages 119–156, 1999. URL `http://citeseer.ist.psu.edu/article/vonoheimb98machinechecking.html`.

J. Wells. Rewriting in the design of type systems. Talk, Sept. 2002. URL `http://www.macs.hw.ac.uk/~jbw/slides/`.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. URL `http://citeseer.ist.psu.edu/wright92syntactic.html`.