A Rewriting Logic Approach to Type Inference: Technical Report

Charles M. Ellison   Traian Serbanuta   Grigore Rosu

March 28, 2008

Abstract

Meseguer and Rosu [MR04, MR07] proposed rewriting logic semantics (RLS) as a programing language definitional framework that unifies operational and algebraic denotational semantics. Once a language is defined as an RLS theory, many generic tools are immediately available for use with no additional cost to the designer. These include a formal inductive theorem proving environment, an efficient interpreter, a state space explorer, and even a model checker. RLS has already been used to define a series of didactic and real languages [MR04, MR07], but its benefits in connection with defining and reasoning about type systems have not been fully investigated yet.

This paper shows how the same RLS style employed for giving formal definitions of languages can be used to define type systems. The same term-rewriting mechanism used to execute RLS language definitions can now be used to execute type systems, giving type checkers or type inferencers. Since both the language and its type system are defined uniformly as theories in the same logic, one can use the standard RLS proof theory to prove properties about languages and type systems for those languages.

The proposed approach is exemplified by defining Milner's polymorphic type inferencer $\mathscr{W}$ as a rewrite logic theory and using the definition: (1) to prove its soundness using Wright and Felleisen's standard preservation and progress methodology [WF94], and (2) to obtain a type inferencer by executing the definition in the Maude rewrite engine. The inferencer obtained "for free" was tested against implementations used in some current functional languages. It was found to be quite competitive—for example, it outperformed SML's type inferencer in all experiments.

To show that the proposed rewriting approach and the resulting type inferencers scale, Milner's simple language is extended with multiple-binding let and letrec, with lists, and with references and side effects. The resulting type inferencer, able to detect weak polymorphism, is only slightly slower than the one for Milner's simpler language. No proofs are given for the extended type system.

We also demonstrate the technique on a number of other languages and type systems, including a simple imperative language, a simply typed, and a polymorphic lambda calculus.

Contents

1 Introduction

This is a preliminary report on using the K Technique for type systems. This document omits numerous proof details, and there are likely many errors in the proof details that are explained. Again, this is supposed to be viewed as a preliminary document and not a finished work by any means.

Rewriting logic has been proposed and used as a semantic foundation for the definition and analysis of languages; see Meseguer and Rosu [MR04, MR07, Ros06] and the references there. Also, in its SOS'05 precursor, [MR07] proposed using the same rewriting logic technique to define type systems and policy checkers for languages; more precisely, to rewrite integer values to their types and to maintain and incrementally rewrite a program until it becomes a type or other desired abstract value. That idea was further explored by Rosu in [Ros06], but, at our knowledge, not used yet to define complex, polymorphic type systems; additionally, [Ros06] provides no implementation, no proofs, and no empirical evaluation of the idea. A similar idea has been recently proposed by Kuan, MacQueen and Findler [KMF07] in the context of Felleisen etal.'s reduction semantics with evaluation contexts [FH92, WF94] and Matthews etal.'s PLT Redex system [MFFF04].

In this paper we show how the same rewriting logic semantics (RLS) framework and definitional style employed in giving formal semantics to languages can be used to also define type systems as rewrite logic theories. This way, both the language and its type system(s) can be defined using the same formalism, facilitating reasoning about programs, languages and type systems. We use Milner's polymorphic type inferencer $\mathscr{W}$ [Mil78] for the EXP language to exemplify our technique. We give one rewrite logic theory for $\mathscr{W}$ and use it both for proving its correctness against a rewrite theory defining EXP and for obtaining an efficient, executable type-inferencer. This paper makes three novel contributions:

1. Shows how non-trivial type systems are defined as RLS theories in a uniform way, following the same style used for defining programming languages and other formal analyses for them;

2. Proposes a type soundness proof technique for languages and type systems defined as RLS theories; and

3. Shows that RLS definitions of type systems, when executed on existing rewrite engines, yield competitive type inferencers.

Related work. In addition to the work mentioned above, there has been other previous work combining term rewriting with type systems. For example, [BKVV05] 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 [Bar91], have been represented in in membership equational logic [SM04], a subset of rewriting logic. There is a large body on term graph rewriting [BvEG$^+$87, Plu98] and its applications to type systems [Ban92, FPST07]. There are similarities with our work, such as using a similar syntax for both types and terms, and a process of reduction or normalization to reduce programs to their types. A collection of theoretical papers on type theory and term rewriting can be found in [KK00]. Adding rewrite rules as annotations to a particular language in order to assist a separate algorithm with type checking has been explored [HdM], as well as adding type annotations to rewrite rules that define program transformations [Mam07]. Much work has been done on defining type systems modularly and proving them sound [LP03, LCH07, KN06]. Much of the recent work in mechanically verified proofs of type soundness has been stimulated by the PoplMark Challenge [ABF$^+$05].

Section 2 introduces RLS and the K definitional style, and gives an RLS definition of Milner's Exp language. Section 3 defines Milner's $\mathscr{W}$ algorithm as an RLS theory and reports on some experiments. Section 4 shows our approach to prove type soundness when both a language and its type system are defined as RLS theories. Section 5 extends the language with lists, references and side effects, showing that our technique scales. Section 6 discusses our implementations and Section 7 concludes the paper.

2 Rewriting Logic, RLS, and K

Term rewriting is a standard computational model supported by many systems. Meseguer's rewriting logic [Mes92], not to be confused with term rewriting, organizes term rewriting modulo equations as a logic with a complete proof system and initial model semantics. Meseguer and Rosu's rewriting logic semantics (RLS) [MR04, MR07] aims at making rewriting logic a foundation for programming language semantics and analysis that unifies operational and algebraic denotational semantics. Rosu's K language definitional style [Ros06] optimizes the use of RLS by means of a definitional technique and a specialized notation. We briefly discuss these and then show how Milner's Exp language [Mil78] can be defined as an RLS theory using K. The rest of the paper employs the same technique to define type systems as RLS theories.

A term rewrite system (TRS) consists of a set of uninterpreted operations over one or more syntactic categories called sorts, and a set of rewrite rules of the form $l \to r$ where l and r are terms that may contain variables. Term rewriting is a method of computation that works by progressively changing (rewriting) a term by using the rules of a TRS. A rule can apply to the entire term being rewritten, or to any subterm of the term. The rewriting process continues as long as it is possible to find a matching subterm. When no matching subterms are found, the rewriting process terminates, with the final term being the result of the computation. Rewriting, like other methods of computation, can continue forever.

There are a large number of term rewriting engines, including ASF [vdBHKO02], Elan [BKK+98], Maude [CDE+02], OBJ [GWM+00], Stratego [Vis03], and others, some of which are capable of executing several million rewrites per second. Rewriting is also a fundamental part of existing languages and theorem provers. Term rewriting is inherently parallel, since non-overlapping parts of a term can be rewritten at the same time, and thus fits well with current trends in system architecture.

In contrast to term rewriting, which is just a method of computation, rewriting logic is a computational logic built upon equational logic, proposed by Meseguer [Mes92] as a logic for true concurrency. In equational logic, a number of *sorts* (types) and *equations* are defined, specifying which terms are equal. Equal terms are members of the same equivalence class; in other words, equal terms are regarded as being identical. Rewriting logic adds *rules* to equational logic, thought of as irreversible transitions: a rewrite theory is an equational theory extended with rewrite rules. Rewriting logic admits a complete proof system and an initial model semantics [Mes92] that makes inductive proofs valid.

Rewriting logic is connected to term rewriting in that all the equations $l = r$ can be transformed into term rewriting rules $l \to r$. This provides a means of taking a rewriting logic theory, together with an initial term, and "executing" it. Any of the exiting rewrite engines can be used for this purpose. Some of the engines, for example Maude [CDE+02], provide even richer support than execution, such as an inductive theorem prover, a state space exploration tool, a model checker, and more.

Rewriting logic semantics (RLS), proposed by Meseguer and Rosu [MR04, MR07], builds upon the observation that programming languages can be defined as rewriting logic theories. By doing so, one gets essentially "for free" not only an interpreter and an initial model semantics for the defined language, but also a series of formal analysis tools obtained as instances of existing tools for rewriting logic. Operationally speaking, the major difference between conventional reduction semantics, with [FH92] or without [Plo04] evaluation contexts, and rewriting logic semantics is that the former typically impose contextual restrictions on applications of reduction steps and the reduction steps happen one at a time, while the latter imposes no such restrictions. To avoid undesired applications of rewrite steps, one has to obey certain methodologies when using rewriting logic. In particular, one can capture the conventional definitional styles by appropriate uses of conditional rules. Consequently, one can define a language many different ways in rewriting logic. In this paper, we use Rosu's K technique [Ros06], which is inspired from abstract state machines [Gur95] and continuations [SW00], and which glosses over many rewriting logic details that are irrelevant for programming languages.

The idea underlying K is to represent the program configuration as a nested "soup" (multiset) of *configuration item* terms representing the current infrastructure needed to process the remaining program or fragment of program; these may include the current computation (a continuation-like structure), environment, store, remaining input, output, analysis results, bookkeeping information, etc. The set of configuration items is not fixed and is typically different from definition to definition. K assumes lists, sets and multisets over any sort whenever needed; for a sort S, $List[S]$ or $SList$ denotes comma-separated lists of terms of sort S, and $Set[S]$ or $SSet$ denotes white space separated sets of terms of sort S. For both lists and sets, we use "$\cdot$" as unit (nil, empty, etc.). If one means a different list- or set-separator, then one writes it as an index; for example, $List_{\curvearrowright}[S]$ stays for $\curvearrowright$-separated lists of terms of sort S. List and sets admit straightforward equational definitions in rewriting logic (list = associative binary operator, set = associative, commutative and idempotent binary operator). Formally, configurations have the following structure:

$$\begin{array}{rcl}
\mathit{ConfigItemLabel} & ::= & \text{(descriptive name, e.g., } k,\ \mathit{env},\ \mathit{store}\text{)} \\
\mathit{ConfigItem} & ::= & \mathit{ConfigItemLabel}(S) \\
 & & (S \text{ can be any sort, including } \mathit{Config}) \\
\mathit{Config} & ::= & \mathit{Set}[\mathit{ConfigItem}]
\end{array}$$

The advantage of representing configurations as nested "soups", is that language rules only need to mention applicable configuration items. This is one aspect of K's modularity. We can add or remove elements from the configuration set as we like, only impacting rules that use those particular items. Rules do not need to be changed to match what the new configuration looks like.

The most important configuration item, present in all K definitions and "wrapped" with the *ConfigItemLabel* k, is the *computation*, denoted by K. Computations generalize abstract syntax by adding a special list construct (associative operator) $\_ \curvearrowright \_$:

$$\begin{array}{rcl} K & ::= & \mathit{KConstant} \mid \mathit{KLabel}(\mathit{List}[K]) \mid \mathit{List}_{\curvearrowright}[K] \\ \mathit{KConstant} & ::= & \text{(one per constant language construct, e.g., } \mathsf{skip}) \\ \mathit{KLabel} & ::= & \text{(one per non-constant language construct)} \\ \mathit{Result} & ::= & \text{(finished computations)} \end{array}$$

The first two constructs for K capture any programming language syntax, under the form of an abstract syntax tree. If one wants more K syntactic categories, then one can do that, too, but we prefer to keep only one here. In our Maude implementation, thanks to Maude's mixfix notation for syntax and implicit reflective capabilities, we actually write programs using the mixfix notation when defining the syntax, and using the K AST-like notation when defining the semantics; in other words, we write "if b then s_1 else s_2" in the syntactic part of the definition and "$\mathsf{if\_then\_else\_}(b, s_1, s_2)$" in the semantics part. Maude's implicit parser will generate the second representation from the first transparently to the user. In other implementations of the K, one may need to use an explicit parser to achieve the same effect. From here on, we take a liberty to interchangeably use either the mixfix or the AST notation for syntax. The distinctive K feature is $\_ \curvearrowright \_$.

Intuitively, $k_1 \curvearrowright k_2$ says "process k_1 then k_2". How this is used and what the meaning of "process" is left open and depends upon the particular definition. For example, in a concrete semantic language definition it can mean "evaluate k_1 then k_2", while in a type inferencer definition it can mean "type and accumulate type constraints in k_1 then in k_2", etc. A K definition consists of two types of sentences: structural equations and rewrite rules. Structural equations carry no computational meaning; they only say which terms should be viewed as identical and their role is to transparently modify the term so that rewrite rules can apply. Rewrite rules are seen as irreversible computational steps and can happen concurrently on a match-and-apply basis. The following are examples of structural equations:

$$\begin{array}{l} a_1 + a_2 = a_1 \curvearrowright \square + a_2 \\ a_1 + a_2 = a_2 \curvearrowright a_1 + \square \\ \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 = b \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \end{array}$$

Note that, unlike in evaluation contexts, $\square$ is not a "hole," but rather part of a *KLabel*, carrying the obvious "plug" intuition; e.g., the *KLabels* involving $\square$ above are $\square + \_$, $\_ + \square$, and $\mathsf{if}\ \square\ \mathsf{then\_else\_}$, respectively. To avoid writing such obvious, distracting, and mechanical structural equations, the author of K proposes to annotate the language syntax with *strict* attributes when defining language constructs: a *strict* construct is associated an equation as above for each of its subexpressions. If an operator is intended to the strict in only some of its arguments, then the positions of the strict arguments are listed as arguments of the *strict* attribute; for example, the above three equations correspond to the attributes *strict* for $\_ + \_$ and *strict*(1) for $\mathsf{if\_then\_else\_}$. We generate all these structural equations automatically from strictness attributes in our implementation (see Section 6).

The following are examples of rewrite rules:

$$\begin{array}{l} i_1 + i_2 \to i, \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \\ \mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \to s_1 \\ \mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \to s_2 \end{array}$$

Structural equations can be applied back and forth; for example, the first equation for $\_ + \_$ can be applied left-to-right to "schedule" a_1 for processing; once evaluated to i_1, the equation is applied in reverse to "plug" the result back in context, then a_2 is scheduled with the second equation left-to-right, then its result i_2 plugged back into context, and then finally the rewrite rule can apply to irreversible apply the computational step. Special care must be taken so that side effects are propagated appropriately: they are only generated at the leftmost side of the computation.

Milner defined and proved the correctness of his $\mathscr{W}$ type inferencer in the context of a simple higher-order language that he called Exp [Mil78]. Recall that $\mathscr{W}$ is the basis for the type checkers of all statically typed functional languages, including ML, OCAML, HASKELL, etc. Exp is a simple expression language containing

$$
\begin{array}{rcll}
\mathit{Var} & ::= & \text{standard identifiers} & \\
\mathit{Exp} & ::= & \mathit{Var} \mid \text{... add basic values (Bools, ints, etc.)} & \\
 & \mid & \lambda\, \mathit{Var}\,.\, \mathit{Exp} & \\
 & \mid & \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
 & \mid & \mu\, \mathit{Var}\,.\, \mathit{Exp} & \\
 & \mid & \textsf{if}\ \mathit{Exp}\ \textsf{then}\ \mathit{Exp}\ \textsf{else}\ \mathit{Exp} & [\mathit{strict}(1)] \\
 & \mid & \textsf{let}\ \mathit{Var} = \mathit{Exp}\ \textsf{in}\ \mathit{Exp} & [\textsf{let}\ x = e\ \textsf{in}\ e' = (\lambda x.e')\, e] \\
 & \mid & \textsf{letrec}\ \mathit{Var}\ \mathit{Var} = \mathit{Exp}\ \textsf{in}\ \mathit{Exp} & \\
 & & \multicolumn{2}{r}{[\textsf{letrec}\ f\ x = e\ \textsf{in}\ e' = \textsf{let}\ f = \mu f.(\lambda x.e)\ \textsf{in}\ e']}
\end{array}
$$

Figure 1: K-Annotated Syntax of Exp

$$
\begin{array}{rcl}
\mathit{Val} & ::= & \lambda\, \mathit{Var}\,.\, \mathit{Exp} \mid \text{...(Bools, ints, etc.)} \\
\mathit{Result} & ::= & \mathit{Val} \\
\mathit{KProper} & ::= & \mu\, \mathit{Var}\,.\, \mathit{Exp} \\
\mathit{ConfigItem} & ::= & k(K) \\
\mathit{Config} & ::= & \mathit{Val} \mid [\![K]\!]_{\mathcal{L}} \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!]_{\mathcal{L}}
\end{array}
$$

$[\![e]\!]_{\mathcal{L}} = [\![k(e)]\!]_{\mathcal{L}}$

$[\![k(v)]\!]_{\mathcal{L}} = v$

$k(\underset{e[x \leftarrow v]}{\underline{(\lambda x.e)\, v}}\rangle$

$k(\underset{e[x \leftarrow \mu\ x.e]}{\underline{\quad \mu\ x.e \quad}}\rangle$

$\textsf{if true then}\ e_1\ \textsf{else}\ e_2 \rightarrow e_1$

$\textsf{if false then}\ e_1\ \textsf{else}\ e_2 \rightarrow e_2$

Figure 2: K Configuration and Semantics of Exp

lambda abstraction and application, conditional, fix point, and "let" and "letrec" binders. To exemplify K and also to remind the reader of Milner's Exp language, we next define it using K. Figure 1 shows its K annotated syntax and Figure 2 shows its K semantics. We also use this to point out some other K notational conventions. Note that application is strict in both its arguments (call-by-value) and that let and letrec are desugared. Additionally, syntactic constructs may be annotated with desugaring equations. In Figure 2, we see that λ-abstractions are defined as values, which are also *Result*s in this definition; *Result*s are not further "scheduled" for processing in structural equations. Since Exp is simple, there is only one *ConfigItem* needed, wrapped by *ConfigItemLabel k*. The first two equations initialize and terminate the computation process. The third applies the β-reduction when $(\lambda x.e)\, v$ is the first item in the computation; we here used two another K pieces of notation: list/set fragment matching and the two-dimensional writing for rules. The first allows us to use angle brackets for unimportant fragments of lists or sets; for example, $(T\rangle$ matches a list whose prefix is T, $\langle T)$ matches a list whose suffix is T, and $\langle T\rangle$ matches a list containing a contiguous fragment T; same for sets, but the three have the same meaning there. Therefore, parentheses represent respective ends of a list/set, while angled brackets mean "the rest". The second allows us to avoid repetition of contexts; for example, instead of writing a rule of the form $C[t_1, t_2, ..., t_n] \rightarrow C[t'_1, t'_2, ..., t'_n]$ (rewriting the above-mentioned subterms in context C) listing the context (which can be quite large) C twice, we can write it $C[\underset{t'_1}{\underline{t_1}}, \underset{t'_2}{\underline{t_2}}, ..., \underset{t'_n}{\underline{t_n}}]$ with the obvious meaning, mentioning the context only once. The remaining Exp semantics is straightforward. Note that we used the conventional substitution, which is also provided in our Maude implementation.

The Exp syntax and semantics defined in Figures 1 and 2 is all we need to write in our implementation of K. To test the semantics, one can now execute programs like the factorial:

```
letrec f x = if x <= 0 then 1 else x * f(x - 1)
in f (f 5)
```

Our definition yields, when run in Maude, an integer of 199 digits in 12,312 rewrites and about 0.2 seconds.

$$\begin{array}{rcll}
\mathit{Var} & ::= & \text{standard identifiers} & \\
\mathit{Exp} & ::= & \mathit{Var} \mid \text{... add basic values (Bools, ints, etc.)} & \\
& \mid & \lambda\, \mathit{Var} \,.\, \mathit{Exp} & \\
& \mid & \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
& \mid & \mu\, \mathit{Var} \,.\, \mathit{Exp} & \\
& \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Exp}\ \mathsf{else}\ \mathit{Exp} & [\mathit{strict}] \\
& \mid & \mathsf{let}\ \mathit{Var} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & [\mathit{strict}(2)] \\
& \mid & \mathsf{letrec}\ \mathit{Var}\ \mathit{Var} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & \\
& & [\mathsf{letrec}\ f\ x = e\ \mathsf{in}\ e' = \mathsf{let}\ f = \mu f.(\lambda x.e)\ \mathsf{in}\ e'] &
\end{array}$$

Figure 3: K-Annotated Syntax of Exp for $\mathscr{W}$

$$\begin{array}{rcl}
\mathit{Type} & ::= & \ldots \mid \mathit{int} \mid \mathit{bool} \mid \mathit{Type} \mapsto \mathit{Type} \mid \mathit{TypeVar} \\
\mathit{Eqn} & ::= & \mathit{Type} = \mathit{Type} \\
\mathit{Eqns} & ::= & \mathit{Set}[\mathit{Eqn}]
\end{array}$$

$$\begin{array}{l}
(t = t) \to \cdot \\
(t_1 \mapsto t_2 = t_1' \mapsto t_2') \to (t_1 = t_1'),\ (t_2 = t_2') \\
(t = t_v) \to (t_v = t) \quad \text{when } t \notin \mathit{TypeVar} \\
t_v = t,\ t_v = t' \to t_v = t,\ t = t' \quad \text{when } t, t' \neq t_v \\
t_v = t,\ t_v' = t' \to t_v = t,\ t_v' = t'[t_v \leftarrow t] \\
\quad \text{when } t_v \neq t_v',\ t_v \neq t,\ t_v' \neq t',\ \text{and } t_v \in \mathit{vars}(t')
\end{array}$$

Figure 4: Unification

3 Defining Milner's $\mathscr{W}$ Type Inferencer

We next define Milner's $\mathscr{W}$ type inferencer [Mil78] using the same K approach. Figure 3 shows the new K annotated syntax for $\mathscr{W}$; it changes the conditional to be strict in all its arguments, and desugars let making it strict in its second argument (because let is typed differently than its desugaring). Unification over type expressions is needed and defined in Figure 4. Fortunately, unification is straightforward to define equationally using set matching; we define it using rewrite rules, though, to emphasize that it is executable ($t_v \in \mathit{TypeVar}$). The first rule in Figure 4 eliminates non-informative type equalities. The second distributes equalities over function types to equalities over their sources and their targets; the third swaps type equalities for convenience (to always have type variables as lhs's of equalities); the fourth ensures that, eventually, no two type equalities have the same lhs variable; finally, the fifth rule canonizes the substitution. As expected, these rules take a set of type equalities and eventually produce a most general unifier for them:

Theorem 1. *Let $\gamma \in \mathit{Eqns}$ be a set of type equations. Then:*

- *The five-rule rewrite system above terminates (modulo AC); let $\theta \in \mathit{Eqns}$ be the normal form of γ.*
- *γ is unifiable iff θ contains only pairs of the form $t_v = t$, where $t_v \notin \mathit{vars}(t)$; if that is the case, then we identify θ with the implicit substitution that it comprises, that is, $\theta(t_v) = t$ when there is some type equality $t_v = t$ in θ, and $\theta(t_v) = t_v$ when there is no type equality of the form $t_v = t$ in θ.*
- *If γ is unifiable then θ is idempotent (i.e., $\theta \circ \theta = \theta$) and is a most general unifier of γ.*

Therefore, the five rules above give us a simple rewriting procedure for unification. The structure of θ in the second item above may be expensive to check every time the unification procedure is invoked; in our Maude implementation of the rules above, we sub-sort (once and for all) each equality of the form $t_v = t$ with $t_v \notin \mathit{vars}(t)$ to a "proper" equality, and then allow only proper equalities in the sort *Eqns* (the improper ones remain part of the "kind" [*Eqns*]). If $\gamma \in \mathit{Eqns}$ is a set of type equations and $t \in \mathit{Type}$ is some type expression, then we let $\gamma[t]$ denote $\theta(t)$; if γ is not unifiable, then $\gamma[t]$ is some error term (in the kind [*Type*] when using Maude).

Figure 5 shows the K definition of $\mathscr{W}$. The configuration has four items: the computation, the type environment, the set of type equations (constraints), and a counter for generating fresh type variables. Due to the strictness attributes, we can assume that the corresponding arguments of the language constructs (in

$$\begin{array}{rcl}
\mathit{Result} & ::= & \mathit{Type} \\
\mathit{TEnv} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Type}] \\
\mathit{Type} & ::= & \ldots \mid \mathit{let}(\mathit{Type}) \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{tenv}(\mathit{TEnv}) \mid \mathit{eqns}(\mathit{Eqns}) \\
 & & \mid \mathit{nextType}(\mathit{TypeVar}) \\
\mathit{Config} & ::= & \mathit{Type} \mid [\![K]\!]_{\mathcal{L}} \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!]_{\mathcal{L}}
\end{array}$$

$$[\![e]\!]_{\mathcal{L}} = [\![k(e)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t_0)]\!]_{\mathcal{L}}$$

$$[\![\langle k(t)\ \mathit{eqns}(\gamma)\rangle]\!]_{\mathcal{L}} = \gamma[t]$$

$i \to \mathit{int},\ \mathit{true} \to \mathit{bool},\ \mathit{false} \to \mathit{bool},$

(and similarly for all the other desired basic values)

$$k(\underset{\mathit{int}}{\underline{t_1 + t_2}}\rangle\ \mathit{eqns}\langle\underset{t_1 = \mathit{int},\ t_2 = \mathit{int}}{\underline{\qquad\cdot\qquad}}\rangle$$

(and similarly for all the other standard operators)

$$k(\underset{(\gamma[t])[tl \leftarrow tl']}{\underline{\qquad x \qquad}}\rangle\ \mathit{tenv}(\eta)\ \mathit{eqns}(\gamma)\ \mathit{nextType}(\underset{t_v + |tl|}{\underline{\quad t_v \quad}})$$

when $\eta[x] = \mathit{let}(t),\ tl = \mathit{vars}(\gamma[t]) - \mathit{vars}(\eta)$
and $tl' = t_v \ldots (t_v + |tl| - 1)$

$$k(\underset{\eta[x]}{\underline{\ x\ }}\rangle\ \mathit{tenv}(\eta) \quad \text{when } \eta[x] \neq \mathit{let}(t)$$

$$k(\underset{(t_v \to e) \curvearrowright \mathit{restore}(\eta)}{\underline{\qquad \lambda x.e \qquad}}\rangle\ \mathit{tenv}(\underset{\eta[x \leftarrow t_v]}{\underline{\quad \eta \quad}})\ \mathit{nextType}(\underset{t_v + 1}{\underline{\ t_v\ }})$$

$$K ::= \cdots \mid \mathit{Type} \to K \quad [\mathit{strict}(2)]$$

$$k(\underset{t_v}{\underline{t_1\ t_2}}\rangle\ \mathit{eqns}\langle\underset{t_1 = t_2 \to t_v}{\underline{\qquad\cdot\qquad}}\rangle\ \mathit{nextType}(\underset{t_v + 1}{\underline{\ t_v\ }})$$

$$k(\underset{e \curvearrowright ?_{=}(t_v) \curvearrowright \mathit{restore}(\eta)}{\underline{\qquad \mu x.e \qquad}}\rangle\ \mathit{tenv}(\underset{\eta[x \leftarrow t_v]}{\underline{\quad \eta \quad}})\ \mathit{nextType}(\underset{t_v + 1}{\underline{\ t_v\ }})$$

$$k(t \to \underset{\cdot}{\underline{?_{=} t_v}}\rangle\ \mathit{eqns}\langle\underset{t_v = t}{\underline{\ \cdot\ }}\rangle$$

$$k(\underset{e \curvearrowright \mathit{restore}(\eta)}{\underline{\ \mathsf{let}\ x = t\ \mathsf{in}\ e\ }}\rangle\ \mathit{env}(\underset{\eta[x \leftarrow \mathit{let}(t)]}{\underline{\qquad \eta \qquad}})$$

$$k(\underset{t_1}{\underline{\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2}}\rangle\ \mathit{eqns}\langle\underset{t = \mathit{bool},\ t_1 = t_2}{\underline{\qquad\cdot\qquad}}\rangle$$

Figure 5: K Configuration and Semantics of $\mathscr{W}$

which these constructs were defined strict) have already been "evaluated" to their types and the corresponding type constraints have been propagated. Lambda and fix-point abstractions perform normal bindings in the type environment, while the let performs a special binding, namely one to a type wrapped with a new "*let*" type construct. When names are looked up in the type environment, the "*let*" types are instantiated with fresh type variables for their "universal" type variables, namely those that do not occur in the type environment.

We believe that the K definition above of $\mathscr{W}$ is as simple, if not simpler, to understand as the original $\mathscr{W}$ procedure proposed by Milner in [Mil78]. However, note that the procedure in [Mil78] is an *algorithm*, almost an *implementation*, rather than a formal definition. Our K definition above is nothing but an ordinary rewriting logic theory, the same as the definition of Exp itself. That should not, and indeed it does not, mean that our K definition, when executed, must necessarily be slower than an actual implementation of $\mathscr{W}$. Experiments using Maude show that our K definition of $\mathscr{W}$ is comparable, or even outperforms, state of the art implementations of type inference in conventional functional languages. In our experiments, it was faster than the type inferencers of SML and Haskell, and only about twice slower than that of OCAML. Concretely,

-	n = 10		n = 11		n = 12		n = 13		n = 14	
OCAML (version 3.09.3)	0.6s	3M	2.2s	2M	8.3s	5M	32.0s	8M	124.9s	13M
Haskell (ghci version 6.8.1)	1.5s	25M	5.2s	28m	21.8s	31M	107.4s	38 M	614.7s	61M
SML (version 110.59)	4.9s	76M	14.7s	110M	111.4s	324M	2129.2s	950M	internal error	
$\mathscr{W}$ in K/Maude2.3 with memo	1.4s	11M	5.9s	33M	23.8s	70M	98.0s	197M	395.9s	653M
$\mathscr{W}$ in K/Maude2.3 without memo	2.5s	10M	7.6s	24M	26.2s	51M	97.5s	156M	367.5s	574M
!$\mathscr{W}$ in K/Maude2.3 with memo	1.4s	12M	5.7s	34M	22.8s	70M	91.9s	198M	377.4s	654M
!$\mathscr{W}$ in K/Maude2.3 without memo	2.4s	11M	7.4s	25M	26.0s	52M	96.9s	156M	359.6s	575M
$\mathscr{W}$ in PLT/Redex[1]	>1h		-		-		-		-	
$\mathscr{W}$ in OCAML	105.9s	1.9M	9m14	2.4M	>1h	2.7M	-		-	

Table 1: Speed of various $\mathscr{W}$ implementations

the program (which is polymorphic in $2^n + 1$ type variables!)

$$
\begin{array}{l}
\mathsf{let}\ f_0 = \lambda x.\lambda y.x\ \mathsf{in} \\
\quad \mathsf{let}\ f_1 = \lambda x.f_0(f_0 x)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ f_2 = \lambda x.f_1(f_1 x)\ \mathsf{in} \\
\quad\quad\quad \ldots \\
\quad\quad\quad\quad \mathsf{let}\ f_n = \lambda x.f_{n-1}(f_{n-1} x)\ \mathsf{in}\ f_n
\end{array}
$$

takes the time/space resources shown in Table 1 to be type checked using OCAML, Haskell, SML, our K definition executed in Maude, and an "off-the-shelf" implementation of $\mathscr{W}$ using OCAML, respectively. These experiments have been conducted on a 3.4GHz/2GB Linux machine. Only the user time has been recorded. Except for SML, the user time was very close to the real time; for SML, the real time was 30% larger than the user time. These ratios appear to scale and are preserved for other programs, too. Moreover, an extension to $\mathscr{W}$, which we call !$\mathscr{W}$, containing lists, products, side effects (through referencing, dereferencing and assignment) and weak polymorphism did not add any noticeable slowdown. Therefore, our K *definitions* surprisingly yield quite realistic *implementations* of type checkers/inferencers when executed on efficient rewrite engines While calculating the numbers above, Maude run at an average of 3 million rewrites per second. In Maude, memoization can be enabled by adding the attribute "`[memo]`" to operations whose results one wants memoized; in our case, we only experimented with memoizing the "built-in" operation `vars : Type -> Set{TypeVar}` in `k-prelude.maude`, which extracts the set of type variables that occur in a type. Memoization appears to pay off when the polymorphic types are small, which is typically the case.

Our Maude "implementation" of an extension[2] of the K definition of $\mathscr{W}$ above has about 30 lines of code. How is it be possible that a formal definition of a type system that one can write in 30 lines of code can be executed *as is* more efficiently than well-engineered implementations of the same type system in widely used programming languages? We think that the answer to this question involves at least two aspects. On the one hand, Maude, despite its generality, is itself a well-engineered rewrite engine implementing state-of-the-art AC matching and term indexing algorithms [Eke96]. On the other hand, our K definition makes intensive use of what Maude is very good at, namely AC matching. For example, note the fourth rule in our rewrite definition of unification[3] that precedes Theorem 1: the type variable t_v appears twice in the lhs of the rule, once in each of the two type equalities involved. Maude will therefore need to search and then index for two type equalities in the set of type constraints which share the same type variable. Similarly, but even more complicatedly, the fifth rule involves two type equalities, the second containing in its t' some occurrence of the type variable t_v that appears in the first. Without appropriate indexing to avoid rematching of rules, which is what Maude does well, such operations can be very expensive. Moreover, note that our type constraints can be "solved" incrementally (by applying the five unification rewrite rules), as generated, into a most general substitution; incremental solving of the type constraints can have a significant impact on the complexity of unification as we defined it, and Maude indeed does that (one can see it by tracing/logging Maude's rewrite steps).

[1]Times for PLT/Redex for lower values of n are as follows: for $n = 7$: 5.0s; for $n = 8$: 181s; and for $n = 9$: 1358.9s
[2]With conventional arithmetic and boolean operators added for writing and testing our definition on meaningful programs
[3]Type unification is "built-in" in K: it is defined as shown above in `k-prelude.maude`.

4 Proofs

Our proofs of type soundness feel very mechanical. There are a number of near-trivial lemmas that must be proved for each of the syntactic constructs of the language, after which the properties of preservation and progress follow.

We use a few conventions to shorten statements. The variables I, V, E, and K stand for integers, values, expressions, and computation items respectively. Additionally, we add $\mathcal{L}$ and $\mathcal{T}$ subscripts on constructs that are shared between both the Exp language and the $\mathscr{W}$ algorithm. 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 $\mathcal{T}$. Finally, for convenience in proofs, our rules are rewritten to match only at the top of the continuation. We can do this without changing the semantics since our rules are orthogonal [Klo92].

We have defined and proved soundness for a number of simple type systems including one for an imperative language, as well as simply typed lambda calculus, lambda calculus with let-polymorphism, and Milner's $\mathscr{W}$ algorithm. Each definition and proof followed the same basic schema, which we outline below:

1. Define a function from Language Configurations to Type Configurations (α)
2. 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
   (b) 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 $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \tau$.
3. Progress: For any expression E where $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R, either $R = V$ for some V, or $\exists R'$ such that $R \longrightarrow R'$.
   (a) Lemma: Inversion of the Typing Relation
   (b) Lemma: Every subterm of a well-typed term is well typed

Further details and more proofs like it can be found in [Ano07a].

4.1 α : $\boldsymbol{Set}[\boldsymbol{ConfigItem}]_{\mathcal{L}} \rightarrow \boldsymbol{Set}[\boldsymbol{ConfigItem}]_{\mathcal{T}}$

A distinguishing feature of our technique is that we use an abstraction function, α, to enable us to convert between a configuration in the language domain to a corresponding configuration in the typing domain. Using an abstraction function in proving soundness is a technique used frequently in the domain of processor construction, as introduced in [HSG98], or compiler optimization [KSK06, KSK07].

Lemma 1. *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. □

We now define α: $\alpha([\![E]\!]_{\mathcal{L}}) = [\![E]\!]_{\mathcal{T}}$

By lemma 1, we know this definition of α is well-defined for all reachable configurations, and homomorphic with respect to structural rules. While the α presented in this paper is, in effect, the identity function, we have experimented with much more complicated αs which lead us to believe the technique scales [Ano07a].

4.2 Preservation and Progress

For the complete proofs of preservation and progress, see section 12.4.

$$
\begin{array}{rcl}
Var & ::= & \text{standard identifiers} \\
Exp & ::= & Var \mid \text{... add basic values (Booleans, integers, etc.)} \\
 & \mid & \mathsf{fun}\, VarList \to Exp \mid Exp\ ExpList \\
 & \mid & \mathsf{let}\ VarList = ExpList\ \mathsf{in}\ Exp \\
 & \mid & \mathsf{letrec}\ VarList = ExpList\ \mathsf{in}\ Exp \\
 & \mid & \mathsf{if}\ Exp\ \mathsf{then}\ Exp\ \mathsf{else}\ Exp \\
 & \mid & \mathsf{ref}\, Exp \mid \&\ Var \mid !\ Exp \mid Exp\ := Exp \\
 & \mid & [ExpList] \mid \mathsf{car}\, Exp \mid \mathsf{cdr}\, Exp \mid \mathsf{null?}\, Exp \\
 & \mid & \mathsf{cons}\, Exp\, Exp \mid Exp\ ;\ Exp
\end{array}
$$

Figure 6: Syntax of $!\mathscr{W}$

5 Extending the type inferencer with lists, references and side effects

In this section we would like to emphasize the fact that, by using rewriting (in the style promoted by this paper), one can easily extend the definition of a type inferencer with little or no impact on the performance. Indeed, as experiments show, time needed for typing programs of the original definitions will remain the same under the new one. This is largely due to the fact that rewriting engines efficiently index rules in such a way that rules orthogonal to the previous ones will add no additional overhead. Moreover, we believe that adding new constructs orthogonally will facilitate reusing the original soundness proof when proving soundness for the extended type system.

We have extended the language Exp by adding lists and references and allowing multiple parameters in functions. The difficulty of typing functional programs in the presence of references is well known and has been intensively studied; e.g., [Tof90, DM82, AMGR93, LW91, Wri95]. In this paper we take a conservative approach, by disallowing programs in which the reference types need to be made weakly polymorphic upon instantiation. For example, we disallow programs like

```
let r = ref list() in r
```

or

```
let f = fun x -> ref x in
    let r = f (fun x -> x) in r
```

because they require the introduction of a weak type variable, but will accept programs like

```
let f = fun x -> ref [] in f
```

because here the reference is protected by a lambda abstraction and not yet activated, and also programs like

```
let f = fun x -> ref x in
    let r = f 3 in r
```

because when determining the type of r, the reference type is instantiated to a concrete type. We chose to reject programs which require weak types rather than defining a mechanism for managing weak type variables, since that seems to be orthogonal to this definition and our goal here is to show that extensions do not have a big negative performance effect for programs written using the original definition.

That is, we allow functions with multiple arguments, let constructs binding multiple variables, references, lists and sequential composition, in addition to the existing definition. For type checker purposes, all operations defined above (except lambda abstractions and lets) are considered strict in all arguments. ref, `!` and `:=` are the standard reference, dereference and assignment operators one can find in functional languages with imperative features and have the same meaning as in ML. `&` gives a pointer to the value held by a variable – for this feature to be possible one has to consider an environment-based memory model for the language.

We only describe the definition of the type inferencer, for the extended language. Since it is an extension of $\mathscr{W}$ with references, let us name it $!\mathscr{W}$. Before defining the type inferencer, we first have to extend the standard set of types with reference and list types, say that they preserve ground types, and finally define a new sort *RefType* for types having only ground reference types unprotected by lambda abstractions. These can be defined as follows:

$$\begin{array}{rcl} \mathit{Type} & ::= & \ldots \mid \mathit{let\ Type} \mid \mathit{ref\ Type} \mid \mathit{list\ Type} \mid \mathit{unit} \\ \mathit{GroundType} & ::= & \ldots \mid \mathit{ref\ GroundType} \mid \mathit{list\ GroundType} \\ \mathit{RefType} & ::= & \mathit{TypeVar} \mid \mathit{GroundType} \mid \mathit{list\ RefType} \\ & \mid & \mathit{TypeList} \to \mathit{Type} \end{array}$$

GroundType is defined in the K prelude together with *Type* and as a subsort of it. *GroundType* is syntactically constrained to only contain types constructed without the use of type variables. Therefore, whenever we extend *Type* with new type constructs, we also need to extend *GroundType* with the corresponding constructs. Types containing only references to ground-types unprotected by abstractions are defined as subsort *RefType* of *Type*, which reduces the check of whether a program should be rejected or not to a membership condition, easily and efficiently implemented into the rewrite engine.

Figure 7 presents the definition of a type inferencer for $!\mathscr{W}$. *bind* and *bindTo* are standard operators for environments/typed environments and are considered part of the K prelude. Specifically, $\mathit{bind}(xl)$ on top of the continuation creates fresh locations (types in this case) for variables xl and assigns them to the variables in the environment. $\mathit{bindTo}(xl)$ expects a list of locations/types on top of it and binds the variables to those locations. *mkFunType* and *mkLet* are standard operators for types, also defined in the K prelude. $\mathit{mkFunType}(xl)$ expects a type t on top of it, looks up the types tl for variables in xl in the environment, producing as output the type $tl \to t$. *mkLet* expects a list of types on top of it and replaces that list of types with the list of corresponding let types, by applying a let type constructor on top of each.

The above definition disallows weak polymorphism by rejecting programs for which weak types would be needed. We are currently researching ways of adding weak polymorphism to this type system without a negative effect on the performance of the inferencer.

6 Implementations

There have been two types of implementations in this project. One consists of an implementation of K in Maude, to facilitate the second type of implementations, namely of a dozen of languages, type systems, and domain-specific dynamic and static checkers. All these are documented and available for download at [Ano07b]. We here can only briefly discuss these, showing only the complete definitions of unification and $\mathscr{W}$.

6.1 Implementing K in Maude

Our implementation consists of one file, `k-prelude.maude`, defining the basic bricks of K: generic lists, sets, maps, computations and configurations. In addition, `k-prelude` also provides language-independent generic features common to many languages, such as: stores (for languages with imperative features), environments, a generic mechanism for generating fresh items (names, locations, types) in the configuration, generic binders and substitution for languages based on substitution, types, type environments and type unification. The `k-prelude` concludes with the definition of an algorithm which uses the reflective capability of Maude to read strictness attributes from an annotated syntax and generate the equations abstracted by them. These definitions are ultimately not Maude-specific; they can be implemented in any language/formalism with support for rewriting. Note, however, that Maude *is* rewriting logic; in other words, using Maude guarantees one that K definitions do not escape the formal universe of rewriting logic, so both efficient executions and formal proofs are possible using the same definition.

`k-prelude` has about 800 lines of Maude code and defines 52 modules. We only show one such module below, namely the one defining type unification, which generalizes the one given in Figure 4 to lists of types, needed for more complex type inferencers, such as the one in Section 5:

```
mod EQNS is including CONFIG + COMMON-TYPE .
 sorts Eqn Eqns . subsort Eqn < Eqns . var Eqn : Eqn .
 var Eqns : Eqns . vars T T' T1 T2 T1' T2' : Type .
 var Tc : TypeConst . vars NeTl NeTl' : NeList{Type} .
```

[4] $\% = \mathit{bind}(xl) \curvearrowright el \curvearrowright \mathit{addEqns}(xl) \curvearrowright \mathit{mkLet}(\cdot) \curvearrowright \mathit{bindTo}(xl)$

$$\begin{array}{rcl}
Result & ::= & Type \\
TEnv & ::= & Map[Name, Type] \\
ConfigItem & ::= & k(K) \mid tenv(TEnv) \mid eqns(Eqns) \\
 & \mid & nextType(TypeVar) \\
Config & ::= & Type \mid [\![K]\!]_{\mathcal{L}} \mid [\![Set[ConfigItem]]\!]_{\mathcal{L}}
\end{array}$$

$[\![e]\!]_{\mathcal{L}} = [\![k(e)\ \ tenv(\cdot)\ \ eqns(\cdot)\ \ nextType(t_v)]\!]_{\mathcal{L}}$

$[\![\langle k(t)\ \ eqns(\gamma)\rangle]\!]_{\mathcal{L}} = \gamma[t]$

$i \to int,\ \ true \to bool,\ \ false \to bool,$

$k(\underset{int}{\underline{t_1 + t_2}}\rangle\ \ eqns\langle\underset{t_1 = int,\ \ t_2 = int}{\underline{\qquad\cdot\qquad}}\rangle$

$k(\underset{t'[tl \leftarrow tl']}{\underline{\qquad x \qquad}}\rangle\ \ tenv(\eta)\ \ eqns(\gamma)\ \ nextType(\underset{t_v + |tl|}{\underline{\quad t_v \quad}})$

when $\eta[x] = let(t),\ \ t' = \gamma[t],\ \ t' : RefType,$

$tl = vars(\gamma[t]) - vars(\eta)$ and $tl' = t_v \ldots (t_v + |tl| - 1)$

$k(\underset{\eta[x]}{\underline{\ x\ }}\rangle\ \ tenv(\eta)$ when $\eta[x] \neq let\ (t)$

$k(\underset{bind(xl) \curvearrowright e \curvearrowright mkFunType(xl) \curvearrowright restore(\eta)}{\underline{\qquad\qquad \lambda xl.e \qquad\qquad}}\rangle\ \ tenv(\eta)$

$k(\underset{t_v}{\underline{t\ tl}}\rangle\ \ eqns\langle\underset{t = tl \to t_v}{\underline{\qquad\cdot\qquad}}\rangle\ \ nextType(\underset{t_v + 1}{\underline{\quad t_v \quad}})$

$k(\underset{el \curvearrowright mkLet(\cdot) \curvearrowright bindTo(xl) \curvearrowright e \curvearrowright restore(\eta)}{\underline{\qquad\qquad \mathsf{let}\, xl = el\ \mathsf{in}\ e \qquad\qquad}}\rangle\ \ tenv(\eta)$

$results(\underset{\cdot}{\underline{t}}\rangle \curvearrowright mkLet\langle\underset{let\ (t)}{\underline{\quad\cdot\quad}})\qquad results(\underset{tl}{\underline{\ \cdot\ }}) \curvearrowright \underset{\cdot}{\underline{mkLet(tl)}}$

$k(\underset{\%^4 \curvearrowright e \curvearrowright restore(\eta)}{\underline{\ \mathsf{letrec}\, xl = el\ \mathsf{in}\ e\ \ }}\rangle tenv(\eta)$

$k(\underset{t_1}{\underline{\mathsf{if}\, t\ \mathsf{then}\, t_1\ \mathsf{else}\, t_2}}\rangle$

$k(\underset{list\ t_v}{\underline{\ [tl]\ }}\rangle\ \ eqns(\Gamma, \underset{t_v\ *\!= tl}{\underline{\qquad\cdot\qquad}})\ \ nextType(\underset{t_v + 1}{\underline{\quad t_v \quad}})$

where $\underset{\cdot}{\underline{t\ *\!= \cdot}}$ and $\underset{t_v = t}{\underline{\qquad\cdot\qquad}}, t_v\ *\!= (\underset{\cdot}{\underline{t}}\rangle$

$k(\underset{list\ t_v}{\underline{\ \mathsf{cdr}\, t\ }}\rangle\ \ eqns(\Gamma, \underset{t = list\ t_v}{\underline{\qquad\cdot\qquad}})\ \ nextType(\underset{t_v + 1}{\underline{\quad t_v \quad}})$

$k(\underset{t_v}{\underline{\mathsf{car}\, t}}\rangle\ \ eqns(\Gamma, \underset{t = list\ t_v}{\underline{\qquad\cdot\qquad}})\ \ nextType(\underset{t_v + 1}{\underline{\quad t_v \quad}})$

$k(\underset{bool}{\underline{\mathsf{null?}\, t}}\rangle\ \ eqns(\Gamma, \underset{t = list\ t_v}{\underline{\qquad\cdot\qquad}})\ \ nextType(\underset{t_v + 1}{\underline{\quad t_v \quad}})$

$k(\underset{t_2}{\underline{\mathsf{cons}\, t_1\, t_2}}\rangle\ \ eqns(\Gamma, \underset{t_2 = list\ t_1}{\underline{\qquad\cdot\qquad}})$

$\mathtt{\&}\ \ t \to ref\, t$

$k(\underset{t_v}{\underline{\mathtt{!}\quad t}}\rangle\ \ eqns\Gamma, \underset{t = ref\, t_v}{\underline{\qquad\cdot\qquad}}\ \ nextType(\underset{t_v + 1}{\underline{\quad t_v \quad}})$

$k(\underset{unit}{\underline{t_1\ \ \mathtt{:=}\ \ t_2}}\rangle\ \ eqns\Gamma, \underset{t_1 = ref\, t_2}{\underline{\qquad\cdot\qquad}}$

$k(\underset{t_2}{\underline{t_1\ \ \mathtt{;}\ \ t_2}}\rangle\ \ eqns\Gamma, \underset{t_1 = unit}{\underline{\qquad\cdot\qquad}}$

Figure 7: K Configuration and Semantics of the $!\mathscr{W}$ type inferencer

```
 vars Tv Tv' : TypeVar . vars Tl Tl' : List{Type} .
 op empty : -> Eqns .
 op _,_ : Eqns Eqns -> Eqns [prec 30 assoc comm id: empty] .
 eq Eqn, Eqn = Eqn .
 op _=_ : List{Type} List{Type} -> [Eqns] [prec 20] .
 eq (nil = nil) = empty .
 eq ((T,NeTl) = (T',NeTl')) = (T = T'), (NeTl = NeTl') .
 eq (T = T) = empty .
 eq (Tl --> T = Tl' --> T') = (Tl = Tl'), (T = T') .
 ceq (T = Tv) = (Tv = T) if not T :: TypeVar .
 cmb Tv = T : Eqn if not(Tv in vars(T)) .
 op eqns_ : Eqns -> ConfigItem .
 eq (Tv = T, Tv = T') = (Tv = T, T = T') .
 ceq (Tv = T, Tv' = T') = (Tv = T, Tv' = T'[Tv <- T])
  if Tv in vars(T') .
 op _[_] : Eqns Type -> Type .
 eq (empty).Eqns[T] = T .
 eq (Tv = T', Eqns)[T] = Eqns[T[Tv <- T']] .
endm
```

Sort *Eqn* stays for one equation and sort *Eqns* for sets of type equations/constraints. The `subsort` keyword is used to say that an equation is identified with the singleton set of equations. After declaring variables, the set constructors are declared: the empty set, and the concatenation/union of sets which is associative, commutative and with identity *empty*. The fact that $\_,\_$ is used to specify sets is emphasized by the idempotency equation $Eqn, Eqn = Eqn$. Constraints are expressed as equalities between lists of types (singleton lists are identified with types). Since solving the constraints may not only lead to a correct substitution, we chose to let constraints be of "error" *kind [Eqns]*. Next two equations say that empty constraints should be discharged and a constraint build form two lists should be transformed into a list of constraints. Next equation if for discharging identity constraints, and the next one for splitting functional constraints into constraints involving their components. An important detail is the membership assertion introduced by the `cmb` keyword, which defines which constraints are acceptable equations: those in which the lhs is a variable not occurring in the rhs. Next equations canonize the constraints and the last two define how a set of constraints applies to a type. Since Maude uses by default an innermost strategy for reduction, one can assume that all previous equations were applied yielding a canonical substitution.

6.2 Defining $\mathscr{W}$ in K-Maude

Figure 9 presents the *complete* definition of W in K-Maude, including both its syntax and semantics. The syntax of *Exp* is annotated with the corresponding strictness attributes, using Maude's `metadata` attribute. `inc` is used to include existing modules. Module `NAME-INT-BOOL-SYNTAX` in `k-prelude` provides definitions for names, integers and booleans, each of them with their corresponding sort, `#Name`, `#Bool`, and `#Int`, respectively. `BINDER-SYNTAX` provides the sort `Exp`, together with a generic binding operation on it: `__._ : Binder #Name Exp -> Exp` and several standard constants of sort binder (e.g., "`lambda`" and "`fix`"). There is a straightforward equivalence between order-sorted signatures in mixfix notation (to which Maude adheres) and context free grammars. Sorts introduce non-terminals, subsorts give non-terminal renamings and operation declarations are grammar productions, the names of the operators giving the terminals of the grammar. E.g., the subsorting introduced by the `subsorts` constraint are translated into the three productions: $Exp ::= ... \mid \#Int \mid \#Bool \mid \#Name$. Similarly,

```
          let_=_in_ : #Name Exp Exp -> Exp
```

precisely corresponds to the CFG production

$$Exp ::= \texttt{let}\ \ \#Name = Exp\ \ \texttt{in}\ \ Exp,$$

introducing the terminals "`let`", "`=`" and "`in`". The keyword `ops` allows us to introduce multiple construct definitions at the same time, provided that they have the same signature/attributes. Since strictness is specific to K, we use the metadata free-form attribute to specify strictness constraints for operations. Attributes like precedences and gathering are used for parsing reasons. This module contains only one equation, desugaring `letrec` to `let` and `fix`.

```
mod EXP-SYNTAX is inc NAME-INT-BOOL-SYNTAX + BINDER-SYNTAX .
 subsorts #Int #Bool #Name < Exp .
 op _+_ : Exp Exp -> Exp
     [gather(E e) prec 33 metadata "strict"] .
 op _*_ : Exp Exp -> Exp
     [gather(E e) prec 31 metadata "strict"] .
 op _<=_ : Exp Exp -> Exp [prec 37 metadata "strict"] .
 op _and_ : Exp Exp -> Exp
     [gather(E e) prec 55 metadata "strict"] .
 op not_ : Exp -> Exp [prec 53 metadata "strict"] .
 op __ : Exp Exp -> Exp
     [prec 10 gather(E e) metadata "strict"] .
 op if_then_else_ : Exp Exp Exp -> Exp [metadata "strict"] .
 op let_=_in_ : #Name Exp Exp -> Exp [metadata "strict(2)"] .
 op letrec__=_in_ : #Name #Name Exp Exp -> Exp .
 vars F X : #Name .  vars E E' : Exp .
 eq (letrec F X = E in E')
  = (let F = fix F . (lambda X . E) in E') .
endm
```

Figure 8: K-Maude definition of $\mathscr{W}$—syntax

The remaining code in Figure 9 defines the structure of the configuration and the semantics of $\mathscr{W}$. In order to increase the modularity of K-Maude by defining generic variants of substitution and code generation, as well as to automatically generate structural equations from strictness attributes, we choose to work with a meta-representation of terms in our current implementation, taking full benefit of the reflective capabilities offered by Maude. When defining the semantics of a language in K-Maude, we first "lift", or "swallow" the syntax of the language into K under the form of ASTs, which are then used to give the actual semantics. The configuration module imports modules for handling the K-ification of AST "leafs" whose syntax is built-in (names, types, etc.) into their semantic counterparts. For example, `#Name`'s semantic correspondent is *Name*. This module references the `META-K` module, which defines syntax for ASTs and computations, together with functions `mkK` and `downK` which allow moving from the syntax level to the AST level and back. Module `COMMON-TENV` provides types, the definition of a typed environment, mapping names to types, together with the definition of the unification presented in Section 6. It also references `CONFIG`, which introduces *Config* and the *k ConfigItemLabel*. `COMMON-META-BINDER` associates the generic syntax definition for binders to their semantic counterpart and gives generic rules for substitution and free variables. The configuration module defines an operation `[[_]]` which starts the evaluation of an expression. Sort *Type* is subsorted into *Config* as it is the resulting sort of the execution. The first equation places the AST of the input expression e into a fresh execution environment: the computation wrapper (k) contains the AST of e, the type environment wrapper (*tenv*) contains the empty environment, the type constraints set wrapper (*eqns*) contains the empty set and the wrapper of the fresh type generator (*nextType*) contains the initial fresh type. The last rule of this module is the one for the end of the execution, when only a type remains in the computation. Then this type, resolved using the accumulated constraints becomes the type of the program. Module `W-EXP-K-SEMANTICS` imports the configuration module, and the automatically generated strictness module. Equations in this module closely and mechanically follow the definition in Figure 5, with syntax being represented in an AST style.

Defining the syntax using Maude's mixfix is a "bonus" that using Maude provides. In principle, one could define directly the semantics, using the mete-representation and then execute the semantics on program given directly in the meta-notation. In other words, the syntax and the semantics are disconnected and can be advanced/improved independently; in particular, one can provide an external parse if one is not happy with Maude's.

7 Conclusions and Further Work

We have shown that rewriting logic, through K, is amenable for defining feasible type inferencers for programming languages and proving type soundness for those definitions. Doing the proof of soundness for

```
mod W-EXP-K-CONFIGURATION is inc EXP-SYNTAX
 + META-NAME-TYPE-DOWN + COMMON-TENV + COMMON-META-BINDER .
 var E : Exp . var T : Type .
 var C : Set{ConfigItem} . var Eqns : Eqns .
 op [[_]] : Exp -> Config .
 subsort Type < Config .
 eq [[E]] = [[k(mkK(E)) tenv(empty)
             eqns(empty) nextType(t(0))]] .
 eq [[k(T) eqns(Eqns) C]] = Eqns[T] .
endm

mod W-EXP-K-SEMANTICS is
 including W-EXP-K-CONFIGURATION + EXP-K-STRICTNESS .
 vars T T' T0 T0' T1 T2 : Type .  var Eqns : Eqns .
 var L : KLabel . vars Tl Tl' : List{Type} .
 var TEnv : Env{Type} .  var X : Name . vars E K : K .
 eq '#int(K) = int . eq '#bool(K) = bool .
--- standard operators
 ceq k(L(T1,T2) -> K) eqns(Eqns)
   = k(int -> K) eqns(Eqns, T1 = int, T2 = int)
  if L == '_+_ or L == '_*_ .
 eq k('_<=_(T1,T2) -> K) eqns(Eqns)
  = k(bool -> K)     eqns(Eqns, T1 = int, T2 = int) .
 eq k('_and_(T1,T2) -> K) eqns(Eqns)
  = k(bool -> K)      eqns(Eqns, T1 = bool, T2 = bool) .
 eq k('not_(T) -> K) eqns(Eqns)
  = k(bool -> K) eqns(Eqns, T = bool) .
--- functional operators
 mb '__._(B:Binder,X,E) : KProper  .
 op let : Type -> Type .
 ceq k(X -> K) tenv(TEnv) eqns(Eqns)nextType(T0)
 = k(T'[Tl <- Tl'] -> K) tenv(TEnv) eqns(Eqns) nextType(T0')
  if let(T) := TEnv[X] /\ T' := Eqns[T]
  /\ Tl := mkList(vars(T') - vars(TEnv))
  /\ T0' := T0 + length(Tl) /\ Tl' := T0 ... (T0' + (-1)) .
 eq k(X -> K)       tenv(TEnv) eqns(Eqns) nextType(T0)
  = k(TEnv[X] -> K) tenv(TEnv) eqns(Eqns) nextType(T0)
   [owise] .
 eq k('__._(binder('lambda),X,E) -> K) tenv(TEnv) nextType(T)
  = k(E -> mkFunType(T) -> restore(TEnv) -> K)
    tenv(TEnv[X <- T]) nextType(T + 1) .
 eq k('__(T1,T2) -> K) eqns(Eqns) nextType(T)
  = k(T -> K) eqns(Eqns, T1 = T2 --> T)nextType(T + 1) .
 eq k('__._(binder('fix),X,E) -> K) tenv(TEnv) nextType(T)
  = k(E -> addEqn(T) -> restore(TEnv) -> K)
    tenv(TEnv[X <- T]) nextType(T + 1) .
 eq k('let_=_in_(X,T,E) -> K)  tenv(TEnv)
  = k(E -> restore(TEnv) -> K) tenv(TEnv[X <- let(T)]) .
 eq k('if_then_else_(T,T1,T2) -> K) eqns(Eqns)
  = k(T1 -> K) eqns(Eqns, T = bool, T1 = T2) .
```

Figure 9: K-Maude definition of $\mathscr{W}$—semantics

$\mathscr{W}$ and other systems using K have led us to believe that this kind of proofs should be easily automatable using Maude's inductive theorem prover. We strongly adhere to the program proposed by the PoplMark Challenge [ABF⁺05], and would like to approach it using the proposed novel methodology.

8 Definition and Proof Schema

1. Language Definition
2. Type System Definition
3. Define a function from Language Configurations to Type Configurations (α)
4. Preservation: If $\llbracket E \rrbracket_{\mathcal{T}} \xrightarrow{*} \tau$ and $\llbracket E \rrbracket_{\mathcal{L}} \xrightarrow{*} V$ for some type τ and value V, then $\llbracket V \rrbracket_{\mathcal{T}} \xrightarrow{*} \tau$.
   (a) Lemma: Structural rules preserve type
   (b) Lemma: If $\llbracket E \rrbracket_{\mathcal{T}} \xrightarrow{*} \tau$ and $\llbracket E \rrbracket_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R, then $\mathcal{T} \models \alpha(R) \xrightarrow{*} \tau$.
   (c) Lemma: If $\mathcal{T} \models \alpha(V) \xrightarrow{*} \tau$ then $\llbracket V \rrbracket_{\mathcal{T}} \xrightarrow{*} \tau$.
5. Progress: For any expression E where $\llbracket E \rrbracket_{\mathcal{T}} \xrightarrow{*} \tau$ and $\llbracket E \rrbracket_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R, either $R = V$ for some V, or $\exists R'$ such that $R \longrightarrow R'$.
   (a) Lemma: Inversion of the Typing Relation
   (b) Lemma: Every subterm of a well-typed term is well typed

9 Simple Imperative Language

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

9.1 $simple_k$ Language

9.1.1 Structural

$$\llbracket \_ \rrbracket_{\mathcal{L}} : \mathit{Exp} \to \mathit{Val} \;\Big\}\cdots\cdots \begin{cases} \underset{k_{\mathcal{L}}(K)\ state(\cdot)}{\underline{\llbracket \quad K \quad \rrbracket}_{\mathcal{L}}} & (1) \\ \llbracket k_{\mathcal{L}}(V)\ state(\_) \rrbracket_{\mathcal{L}} \equiv V & (2) \end{cases}$$

$$\_[\_] : \mathit{State} \times \mathit{Var} \to \mathit{Int} \;\Big\}\cdots\cdots \begin{cases} k_{\mathcal{L}}(\underset{S[X]}{\underline{X}} \rangle\ state(S) & (3) \end{cases}$$

$$\left.\begin{array}{l} \underline{\_+\_} : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \\ \underline{\mathsf{not}\ \_} : \mathit{Exp} \to \mathit{Exp} \\ \underline{\_\ \mathsf{and}\ \_} : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \\ \underline{\_<\_} : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \\ \underline{\_{:=}\_} : \mathit{Var} \times \mathit{Exp} \to \mathit{Exp} \end{array}\right\}\cdots\cdots \begin{cases} k_{\mathcal{L}}(\underset{K \curvearrowright \square + K'}{\underline{K + K'}} \rangle & (4) \\ k_{\mathcal{L}}(\underset{K' \curvearrowright K + \square}{\underline{K + K'}} \rangle & (5) \\ k_{\mathcal{L}}(\underset{K \curvearrowright \mathsf{not}\ \square}{\underline{\mathsf{not}\ K}} \rangle & (6) \\ k_{\mathcal{L}}(\underset{K \curvearrowright \square\ \mathsf{and}\ K'}{\underline{K\ \mathsf{and}\ K'}} \rangle & (7) \\ k_{\mathcal{L}}(\underset{K' \curvearrowright K\ \mathsf{and}\ \square}{\underline{K\ \mathsf{and}\ K'}} \rangle & (8) \\ k_{\mathcal{L}}(\underset{K \curvearrowright \square < K'}{\underline{K < K'}} \rangle & (9) \\ k_{\mathcal{L}}(\underset{K' \curvearrowright K < \square}{\underline{K < K'}} \rangle & (10) \\ k_{\mathcal{L}}(\underset{K \curvearrowright X{:=}\square}{\underline{X{:=}K}} \rangle & (11) \end{cases}$$

$$\left.\begin{array}{l}\underline{\_\,;\,\_} \;:\; \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \\ \underline{\textsf{if}\ \_\ \textsf{then}\ \_\ \textsf{else}\ \_} \;:\; \mathit{Exp} \times \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \\ \underline{\textsf{while}\ \_\ \textsf{do}\ \_} \;:\; \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}\end{array}\right\}\cdot\cdot\left\{\begin{array}{lr} k_{\mathcal{L}}(\dfrac{K\,;K'}{K \curvearrowright \Box\,;K'}\rangle & (12) \\ k_{\mathcal{L}}(\dfrac{\textsf{if}\ K\ \textsf{then}\ S\ \textsf{else}\ S'}{K \curvearrowright \textsf{if}\ \Box\ \textsf{then}\ S\ \textsf{else}\ S'}\rangle & (13) \\ k_{\mathcal{L}}(\dfrac{\textsf{while}\ K\ \textsf{do}\ K'}{\textsf{if}\ K\ \textsf{then}\ (K'\,;\textsf{while}\ K\ \textsf{do}\ K')\ \textsf{else}\ \textsf{skip}}\rangle & (14)\end{array}\right.$$

9.1.2 Semantic

$$\left.\begin{array}{l}\_[\_ \leftarrow \_] \;:\; \mathit{State} \times \mathit{Var} \times \mathit{Int} \to \mathit{State} \\ \underline{\textsf{true}} \;:\; \to \mathit{Val} \\ \underline{\textsf{false}} \;:\; \to \mathit{Val} \\ \underline{\textsf{skip}} \;:\; \to \mathit{Exp}\end{array}\right\}\cdots\cdots\left\{\begin{array}{lr} k_{\mathcal{L}}(\dfrac{(([X,I]\ \_)[X]}{I}\rangle & (15) \\ k_{\mathcal{L}}(\dfrac{S[X]}{0}\rangle \text{ if } X \notin S & (16) \\ (([X,\_]\ S)[X \leftarrow I] \longrightarrow [X,I]\ S & (17) \\ S[X \leftarrow I] \longrightarrow [X,I]\ S \text{ if } X \notin S & (18) \\ k_{\mathcal{L}}(\dfrac{I + I'}{I +_{int} I'}\rangle & (19) \\ k_{\mathcal{L}}(\dfrac{B\ \textsf{and}\ B'}{B\ and_{bool}\ B'}\rangle & (20) \\ k_{\mathcal{L}}(\dfrac{\textsf{not}\ B}{not_{bool}\ B}\rangle & (21) \\ k_{\mathcal{L}}(\dfrac{I < I'}{I <_{int} I'}\rangle & (22) \\ k_{\mathcal{L}}(\dfrac{\textsf{skip}\,;K}{K}\rangle & (23) \\ k(\dfrac{X{:=}I}{\textsf{skip}}\rangle\ \mathit{state}(\dfrac{S}{S[X \leftarrow I]}) & (24) \\ k_{\mathcal{L}}(\dfrac{\textsf{if true then}\ S\ \textsf{else}\ S'}{S}\rangle & (25) \\ k_{\mathcal{L}}(\dfrac{\textsf{if false then}\ S\ \textsf{else}\ S'}{S'}\rangle & (26)\end{array}\right.$$

9.2 $simple_k$ Type Checker

9.2.1 Structural

$$\llbracket \_ \rrbracket_{\mathcal{T}} \;:\; \mathit{Exp} \to \mathit{Type} \;\Big\}\cdots\cdots\cdots\cdots\cdots\cdots\left\{\begin{array}{lr} \llbracket \dfrac{E}{k_{\mathcal{T}}(E)} \rrbracket_{\mathcal{T}} & (27) \\ \llbracket k_{\mathcal{T}}(\tau) \rrbracket_{\mathcal{T}} \equiv \tau & (28)\end{array}\right.$$

9.2.2 Semantic

$$\left.\begin{array}{l} \_[\_ \leftarrow \_] \ : \ \mathit{State} \times \mathit{Var} \times \mathit{Int} \rightarrow \mathit{State} \\ \underline{\mathsf{true}} \ : \ \rightarrow \mathit{Val} \\ \underline{\mathsf{false}} \ : \ \rightarrow \mathit{Val} \\ \underline{\mathsf{skip}} \ : \ \rightarrow \mathit{Exp} \\ \textsc{Int} \ : \ \rightarrow \mathit{Type} \\ \textsc{Bool} \ : \ \rightarrow \mathit{Type} \\ \textsc{Statement} \ : \ \rightarrow \mathit{Type} \end{array}\right\} \cdots\cdots \left\{\begin{array}{rll} V : \mathit{Int} & \longrightarrow \textsc{Int} & (29) \\ B : \mathit{Bool} & \longrightarrow \textsc{Bool} & (30) \\ \mathsf{skip} & \longrightarrow \textsc{Statement} & (31) \\ S[X] & \longrightarrow \textsc{Int} & (32) \\ \textsc{Int} + \textsc{Int} & \longrightarrow \textsc{Int} & (33) \\ \textsc{Bool} \ \mathsf{and} \ \textsc{Bool} & \longrightarrow \textsc{Bool} & (34) \\ \mathsf{not} \ \textsc{Bool} & \longrightarrow \textsc{Bool} & (35) \\ \textsc{Int} < \textsc{Int} & \longrightarrow \textsc{Bool} & (36) \\ \textsc{Statement} \ ; K & \longrightarrow K & (37) \\ X{:=}\textsc{Int} & \longrightarrow \textsc{Statement} & (38) \\ \mathsf{if} \ \textsc{Bool} \ \mathsf{then} \ \tau \ \mathsf{else} \ \tau & \longrightarrow \tau & (39) \\ \mathsf{while} \ \textsc{Bool} \ \mathsf{do} \ \textsc{Statement} & \longrightarrow \textsc{Statement} & (40) \end{array}\right.$$

9.3 *simple*$_k$ α

$$\alpha \ : \ \mathit{State}_{\mathcal{L}} \rightarrow \mathit{State}_{\mathcal{T}} \ \Big\} \cdots\cdots \left\{\begin{array}{rl} \alpha([\![E]\!]_{\mathcal{L}}) = [\![E]\!]_{\mathcal{T}} & (41) \\ \alpha([\![k_{\mathcal{L}}(K) \ \mathit{state}(\_)]\!]_{\mathcal{L}}) = [\![k_{\mathcal{T}}(K)]\!]_{\mathcal{T}} & (42) \end{array}\right.$$

$$\alpha \ : \ K_{\mathcal{L}} \rightarrow K_{\mathcal{T}} \ \Big\} \cdots\cdots \Big\{ \ \alpha(K) = [\![K]\!]_{\mathcal{T}} \quad (43)$$

9.4 *simple*$_k$ Proofs

9.4.1 Preservation

Lemma 2. *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 41, 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.

Rule 2 $R = [\![k_{\mathcal{L}}(V) \ \mathit{state}(\_)]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}(V)]\!]_{\mathcal{T}}$	42
$[\![V]\!]_{\mathcal{T}}$	27

$R' = V$

$\alpha(R')$	By Rule
$[\![V]\!]_{\mathcal{T}}$	43

Rule 19 $R = [\![k_{\mathcal{L}}(I : \mathit{Int} + I' : \mathit{Int} \curvearrowright K) \ \mathit{state}(\_)]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}(I + I' \curvearrowright K)]\!]_{\mathcal{T}}$	42
$[\![k_{\mathcal{T}}(\textsc{Int} + I' \curvearrowright K)]\!]_{\mathcal{T}}$	29
$[\![k_{\mathcal{T}}(\textsc{Int} + \textsc{Int} \curvearrowright K)]\!]_{\mathcal{T}}$	29
$[\![k_{\mathcal{T}}(\textsc{Int} \curvearrowright K)]\!]_{\mathcal{T}}$	33

$R' = [\![k_{\mathcal{L}}(I +_{int} I' \curvearrowright K) \ \mathit{state}(\_)]\!]_{\mathcal{L}}$

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(I +_{int} I' \curvearrowright K)]\!]_{\mathcal{T}}$	42
$[\![k_{\mathcal{T}}(\textsc{Int} \curvearrowright K)]\!]_{\mathcal{T}}$	29

Rule 24 $R = [\![k_{\mathcal{L}}(X{:=}I : \mathit{Int} \curvearrowright K)\ \mathit{state}(S)]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}(X{:=}I \curvearrowright K)]\!]_{\mathcal{T}}$	42
$[\![k_{\mathcal{T}}(X{:=}\textsc{Int} \curvearrowright K)]\!]_{\mathcal{T}}$	29
$[\![k_{\mathcal{T}}(\textsc{Statement} \curvearrowright K)]\!]_{\mathcal{T}}$	38

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

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(\mathsf{skip} \curvearrowright K)]\!]_{\mathcal{T}}$	42
$[\![k_{\mathcal{T}}(\textsc{Statement} \curvearrowright K)]\!]_{\mathcal{T}}$	31

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

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}(\mathsf{if\ true\ then}\ S\ \mathsf{else}\ S' \curvearrowright K)]\!]_{\mathcal{T}}$	42
$[\![k_{\mathcal{T}}(\mathsf{if\ true\ then}\ \tau\ \mathsf{else}\ S' \curvearrowright K)]\!]_{\mathcal{T}}$	assm.
$[\![k_{\mathcal{T}}(\mathsf{if\ true\ then}\ \tau\ \mathsf{else}\ \tau \curvearrowright K)]\!]_{\mathcal{T}}$	assm.
$[\![k_{\mathcal{T}}(\tau \curvearrowright K)]\!]_{\mathcal{T}}$	39

$R' = [\![k_{\mathcal{L}}(S \curvearrowright K)\ \mathit{state}(\_)]\!]_{\mathcal{L}}$

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(S \curvearrowright K)]\!]_{\mathcal{T}}$	42
$[\![k_{\mathcal{T}}(\tau \curvearrowright K)]\!]_{\mathcal{T}}$	assm. & topOfStack lemma

This actually isn't correct since we don't know whether reducing a type alone on the stack is the same as reducing it within a context. Easiest fix is to add $k_{\mathcal{T}}$ to the front of all the typing rules.

Rule 14 $R = [\![k_{\mathcal{L}}(\mathsf{while}\ K\ \mathsf{do}\ K' \curvearrowright J)\ \mathit{state}(\_)]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}(\mathsf{while}\ K\ \mathsf{do}\ K' \curvearrowright J)]\!]_{\mathcal{T}}$	42
$[\![k_{\mathcal{T}}(\textsc{Statement} \curvearrowright J)]\!]_{\mathcal{T}}$	assm. & 40

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

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(\mathsf{if}\ K\ \mathsf{then}\ (K'\,;\mathsf{while}\ K\ \mathsf{do}\ K')\ \mathsf{else\ skip} \curvearrowright J)]\!]_{\mathcal{T}}$	42
$[\![k_{\mathcal{T}}(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ (\textsc{Statement}\,;\mathsf{while}\ \textsc{Bool}\ \mathsf{do}\ \textsc{Statement})\ \mathsf{else\ skip} \curvearrowright J)]\!]_{\mathcal{T}}$	assm. & 40
$[\![k_{\mathcal{T}}(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ (\textsc{Statement}\,;\textsc{Statement})\ \mathsf{else\ skip} \curvearrowright J)]\!]_{\mathcal{T}}$	40
$[\![k_{\mathcal{T}}(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \textsc{Statement}\ \mathsf{else\ skip} \curvearrowright J)]\!]_{\mathcal{T}}$	37
$[\![k_{\mathcal{T}}(\mathsf{if}\ \textsc{Bool}\ \mathsf{then}\ \textsc{Statement}\ \mathsf{else}\ \textsc{Statement} \curvearrowright J)]\!]_{\mathcal{T}}$	31
$[\![k_{\mathcal{T}}(\textsc{Statement} \curvearrowright J)]\!]_{\mathcal{T}}$	39

□

9.4.2 Progress

Lemma 3. *All expressions on the stack are either the only thing on the stack, or immediately preceding an expression with a hole.*

Lemma 4. *If $E_1 + E_2$ is an expression such that $[\![E_1 + E_2]\!]_{\mathcal{T}} \xrightarrow{*} \tau$, then $[\![E_1]\!]_{\mathcal{T}} \xrightarrow{*} \textsc{Int}$ and $[\![E_2]\!]_{\mathcal{T}} \xrightarrow{*} \textsc{Int}$*

Lemma 5. *If $E_1 + E_2$ is an expression such that $[\![E_1 + E_2]\!]_{\mathcal{T}} \xrightarrow{*} \tau$, then if $[\![E_1]\!]_{\mathcal{L}} \xrightarrow{*} V : \mathit{Val}$, we have that $[\![V]\!]_{\mathcal{T}} \xrightarrow{*} \textsc{Int}$.*

Proof. This follows by Preservation and lemma 4. □

Lemma 6. *If E is an expression such that $\llbracket E \rrbracket_{\mathcal{T}} \xrightarrow{*} \tau$, then if $\mathcal{L} \models \{\!\!\{\ k(E \curvearrowright K)\ state(S)\ \}\!\!\} \xrightarrow{*} \{\!\!\{\ k(V : Val \curvearrowright K)\ state(S')\ \}\!\!\}$ for some K, S, and S', then $V : \tau$.*

Lemma 7. *Let E be an expression such that $\llbracket E \rrbracket_{\mathcal{T}} = \tau$. Then for any sub-expression S of E, if $\mathcal{L} \models \{\!\!\{\ k(S \curvearrowright K)\ state(S)\ \}\!\!\} \xrightarrow{*} \{\!\!\{\ k(V : Val \curvearrowright K)\ state(S')\ \}\!\!\}$ for some K, S, and S', then V has the expected type.*

Lemma 8. *Any reachable configuration can be transformed using structural rules into an expression.*

Lemma 9 (Progress)**.** *Let E be an expression such that $\llbracket E \rrbracket_{\mathcal{T}} = \tau$ and $\llbracket E \rrbracket_{\mathcal{L}} \xrightarrow{*} R$ for some R. Either $R = V : Val$ for some V, or $\exists R'$ such that $R \longrightarrow R'$.*

Proof. Let E be an expression such that $\llbracket E \rrbracket_{\mathcal{T}} = \tau$ and $\llbracket E \rrbracket_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R.

Induction on the size of the expression for R implied by lemma 8. If there are no operators, then the expression is a value and we are done. If there is one operator then the key subexpressions must all be values, and by lemma 7 we know we can apply the appropriate rule to reduce the expression.

Now in the inductive case, assume all expressions of size less than n that type are either values or can take a step. Consider an expression of size n that types. If all of its key subexpressions are values, then we can apply one of the language rules on the expression directly. If not, then there must be a key subexpression that is not a value, so we can move it in front using a structural rule, then by inductive assumption, we know we can take a step on that expression. ☐

10 Monomorphic λ Calculus

10.1 λ Calculus Language

10.1.1 Structural

$$\left.\begin{array}{l} \llbracket \_ \rrbracket_{\mathcal{L}}\ :\ Exp \to Val \\ \llbracket \_ \rrbracket_{\mathcal{L}}\ :\ Set[ConfigItem] \to Val \\ k_{\mathcal{L}}(\_)\ :\ K \to Set[ConfigItem] \end{array}\right\} \cdots\cdots\cdots\cdots \left\{\begin{array}{rr} \dfrac{\llbracket\ E\ \rrbracket_{\mathcal{L}}}{k_{\mathcal{L}}(E)} & (44) \\ \llbracket k_{\mathcal{L}}(V) \rrbracket_{\mathcal{L}} \equiv V & (45) \\ K K' \equiv K \curvearrowright \square K' & (46) \\ K K' \equiv K' \curvearrowright K \square & (47) \end{array}\right.$$

10.1.2 Semantic

$$\left.\begin{array}{l} \underline{\lambda \_ : \_ . \_}\ :\ Var \times Type \times Exp \to Val \\ \underline{\_\ \_}\ :\ Exp \times Exp \to Exp \\ \_[\_/\_]\ :\ Exp \times Exp \times Var \to Exp \end{array}\right\} \cdots\cdots\cdots\cdots \left\{\begin{array}{rr} k_{\mathcal{L}}(\underset{E[V/X]}{\underline{(\lambda X : \tau \,.\, E)(V : Val)}}\rangle & (48) \end{array}\right.$$

10.2 λ Calculus Type Checker

10.2.1 Structural

$$\left.\begin{array}{l} \llbracket \_ \rrbracket_{\mathcal{T}}\ :\ Exp \to Type \\ \llbracket \_ \rrbracket_{\mathcal{T}}\ :\ Set[ConfigItem] \to Type \\ k_{\mathcal{T}}(\_)\ :\ K \to Set[ConfigItem] \end{array}\right\} \cdots\cdots\cdots\cdots \left\{\begin{array}{rr} \dfrac{\llbracket\ E\ \rrbracket_{\mathcal{T}}}{k_{\mathcal{T}}(E)} & (49) \\ \llbracket k_{\mathcal{T}}(\tau) \rrbracket_{\mathcal{T}} \equiv \tau & (50) \\ K K' \equiv K \curvearrowright \square K' & (51) \\ K K' \equiv K' \curvearrowright K \square & (52) \\ \tau \to K \equiv K \curvearrowright \tau \to \square & (53) \end{array}\right.$$

10.2.2 Semantic

$$\left.\begin{array}{l} \bullet\ :\ \to Type \\ \_ \to \_\ :\ Type \times Type \to Type \end{array}\right\} \cdots\cdots\cdots\cdots \left\{\begin{array}{lr} \lambda X : \tau \,.\, K \longrightarrow \tau \to K[\tau/X] & (54) \\ (\tau \to \tau')(\tau) \longrightarrow \tau' & (55) \end{array}\right.$$

10.3 λ Calculus α

$$\alpha \;:\; \mathit{Result} \to \mathit{Result} \;\Big\}\cdots\cdots \begin{cases} \alpha([\![E]\!]_{\mathcal{L}}) = [\![E]\!]_{\mathcal{T}} & (56)\\ \alpha([\![k_{\mathcal{L}}(K)]\!]_{\mathcal{L}}) = [\![k_{\mathcal{T}}(K)]\!]_{\mathcal{T}} & (57)\\ \alpha(V : \mathit{Val}) = [\![V]\!]_{\mathcal{T}} & (58)\end{cases}$$

10.4 λ Calculus Proofs

Lemma 10. *Alpha is a total function over all reachable configurations of the language, modulo the structural rules of the language.*

Lemma 11. *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 12. *The structural rules of the type system preserve type.*

Lemma 13. *If the language rules or the typing rules reduce an expression to a form where only structural rules apply, then that form is unique up to the structural rules.*

Lemma 14. *If $[\![k_{\mathcal{T}}(E[\tau'/X] \curvearrowright K)]\!]_{\mathcal{T}} \xrightarrow{*} [\![k_{\mathcal{T}}(\tau \curvearrowright K)]\!]_{\mathcal{T}}$ and $[\![k_{\mathcal{T}}(V \curvearrowright K')]\!]_{\mathcal{T}} \xrightarrow{*} [\![k_{\mathcal{T}}(\tau' \curvearrowright K')]\!]_{\mathcal{T}}$, then $[\![k_{\mathcal{T}}(E[V/X] \curvearrowright K)]\!]_{\mathcal{T}} \xrightarrow{*} [\![k_{\mathcal{T}}(\tau \curvearrowright K)]\!]_{\mathcal{T}}$.*

Proof. Assume $[\![k_{\mathcal{T}}(E[\tau'/X] \curvearrowright K)]\!]_{\mathcal{T}} \xrightarrow{*} [\![k_{\mathcal{T}}(\tau \curvearrowright K)]\!]_{\mathcal{T}}$ and $[\![k_{\mathcal{T}}(V \curvearrowright K')]\!]_{\mathcal{T}} \xrightarrow{*} [\![k_{\mathcal{T}}(\tau' \curvearrowright K')]\!]_{\mathcal{T}}$ 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 $[\![k_{\mathcal{T}}(E[\tau'/X] \curvearrowright K)]\!]_{\mathcal{T}} = [\![k_{\mathcal{T}}(\tau' \curvearrowright K)]\!]_{\mathcal{T}}$ and $[\![k_{\mathcal{T}}(E[V/X] \curvearrowright K)]\!]_{\mathcal{T}} = [\![k_{\mathcal{T}}(V \curvearrowright K)]\!]_{\mathcal{T}}$ which by assumption reduces to $[\![k_{\mathcal{T}}(\tau' \curvearrowright K)]\!]_{\mathcal{T}}$. We see that the property holds in this case.

Alternatively, if $E = Y$ where $Y \neq X$ then $[\![k_{\mathcal{T}}(E[\tau'/X] \curvearrowright K)]\!]_{\mathcal{T}} = [\![k_{\mathcal{T}}(Y \curvearrowright K)]\!]_{\mathcal{T}}$. 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 that $[\![k_{\mathcal{T}}(E[\tau'/X] \curvearrowright K)]\!]_{\mathcal{T}} = [\![k_{\mathcal{T}}((\lambda Y{:}\sigma\,.\,E')[\tau'/X] \curvearrowright K)]\!]_{\mathcal{T}}$. Using the typing rule and properties of substitution, we see that this reduces to $[\![k_{\mathcal{T}}(\sigma \to (E'[\sigma/Y])[\tau'/X] \curvearrowright K)]\!]_{\mathcal{T}}$. Finally, this is equivalent to $[\![k_{\mathcal{T}}((E'[\sigma/Y])[\tau'/X] \curvearrowright \sigma \to \square \curvearrowright K)]\!]_{\mathcal{T}}$ by the structural rule.

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

If it is an application expression, ...

□

10.4.1 Preservation

Lemma 15 (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 56 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 44 $R = [\![E]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![E]\!]_{\mathcal{T}}$	56

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

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(E)]\!]_{\mathcal{T}}$	57
$[\![E]\!]_{\mathcal{T}}$	49

Rule 45 $R = [\![k_{\mathcal{L}}(V)]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}(V)]\!]_{\mathcal{T}}$	57

$R' = V$

$\alpha(R')$	By Rule
$[\![V]\!]_{\mathcal{T}}$	58
$[\![k_{\mathcal{T}}(V)]\!]_{\mathcal{T}}$	50

Rules 46 and 47 These follow because the type system has corresponding structural rules 51 and 52.

Rule 48 $R = [\![k_{\mathcal{L}}((\lambda X : \tau' . E)(V : \mathit{Val}) \curvearrowright K)]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}((\lambda X : \tau' . E)(V : \mathit{Val}) \curvearrowright K)]\!]_{\mathcal{T}}$	57
$[\![k_{\mathcal{T}}((\tau' \to E[\tau'/X])(V : \mathit{Val}) \curvearrowright K)]\!]_{\mathcal{T}}$	54
$[\![k_{\mathcal{T}}((\tau' \to E[\tau'/X])(\tau') \curvearrowright K)]\!]_{\mathcal{T}}$	Assm.
$[\![k_{\mathcal{T}}(E[\tau'/X] \curvearrowright K)]\!]_{\mathcal{T}}$	55

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

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(E[V/X] \curvearrowright K)]\!]_{\mathcal{T}}$	57
$[\![k_{\mathcal{T}}(E[\tau'/X] \curvearrowright K)]\!]_{\mathcal{T}}$	Above & Lemma 14

□

10.4.2 Progress

Lemma 16 (Progress)**.** *Let E be an expression such that $[\![E]\!]_{\mathcal{T}} = \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some R. Either $R = V : \mathit{Val}$ for some V, or $\exists R'$ such that $R \longrightarrow R'$.*

Proof. Let E be an expression such that $[\![E]\!]_{\mathcal{T}} = \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R. We know by lemma 11 that any such R is equivalent under the structural rules to something of the form $[\![E']\!]_{\mathcal{L}}$ and therefore $[\![k_{\mathcal{L}}(E')]\!]_{\mathcal{L}}$ for some expression E'. This expression can be one of a few forms, namely, a variable, a lambda expression, or an application expression. We consider each individually.

($\boldsymbol{X}$: ***Var***) If E' is just a variable name, we know by preservation that this should type to τ, but there is no rule allowing us to type variables. Therefore, this case cannot arise.

($\lambda \boldsymbol{X} : \tau' . E''$) In this case, E' is a value so the property holds.

(E'')(E''') We know by preservation that this should type to τ. The only way that is true is if E'' is a lambda term or if some subexpression of E'' is an application with a lambda expression on the left. In either case, we know we can take the corresponding step in the language.

We see that in all cases the property holds. □

11 Poly λ Calculus

In this section we present the full polymorphic lambda calculus with constants as described in [WF94]. 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.

11.1 Poly λ Calculus Language

11.1.1 Structural

$$\left.\begin{array}{l}[\![\_]\!]_{\mathcal{L}} \;:\; \mathit{Exp} \to \mathit{Val}\\ [\![\_]\!]_{\mathcal{L}} \;:\; \mathit{Set}[\mathit{ConfigItem}] \to \mathit{Val}\\ k_{\mathcal{L}}(\_) \;:\; K \to \mathit{Set}[\mathit{ConfigItem}]\end{array}\right\}\cdots\cdots\left\{\begin{array}{lr}[\![\frac{E}{k_{\mathcal{L}}(E)}]\!]_{\mathcal{L}} & (59)\\ [\![k_{\mathcal{L}}(V)]\!]_{\mathcal{L}} \equiv V & (60)\\ EE' \equiv E \curvearrowright \Box E' & (61)\\ (V : \mathit{Val})E \equiv E \curvearrowright V\Box & (62)\\ \mathsf{let}\,X\ \mathsf{be}\ E\ \mathsf{in}\ E' \equiv E \curvearrowright \mathsf{let}\,X\ \mathsf{be}\ \Box\ \mathsf{in}\ E' & (63)\end{array}\right.$$

11.1.2 Semantic

$$\left.\begin{array}{l}\underline{\lambda\_.\_} \;:\; \mathit{Var}\times\mathit{Exp}\to\mathit{Val}\\ \underline{\mathsf{let}\ \_\ \mathsf{be}\ \_\ \mathsf{in}\ \_} \;:\; \mathit{Var}\times\mathit{Exp}\times\mathit{Exp}\to\mathit{Exp}\\ \underline{\_\ \_} \;:\; \mathit{Exp}\times\mathit{Exp}\to\mathit{Exp}\\ \underline{\mathsf{Y}} \;:\; \to\mathit{Val}\\ \delta(\_,\_) \;:\; \mathit{Const}\times\mathit{ClosedVal}\to\mathit{ClosedVal}\\ \_[\_/\_] \;:\; \mathit{Exp}\times\mathit{Exp}\times\mathit{Var}\to\mathit{Exp}\end{array}\right\}\cdots\cdots\left\{\begin{array}{lr}k_{\mathcal{L}}(\frac{(\lambda X\,.\,E)(V:\mathit{Val})}{E[V/X]}\rangle & (64)\\ k_{\mathcal{L}}(\frac{\mathsf{let}\,X\ \mathsf{be}\ V:\mathit{Val}\ \mathsf{in}\ E}{E[V/X]}\rangle & (65)\\ k_{\mathcal{L}}(\frac{(C:\mathit{Const})(V:\mathit{Val})}{\delta(C,V)}\rangle & (66)\\ k_{\mathcal{L}}(\frac{\mathsf{Y}(V:\mathit{Val})}{V(\lambda X\,.\,(\mathsf{Y}V)X)}\rangle & (67)\end{array}\right.$$

11.2 Poly λ Calculus Type Inferencer

11.2.1 Structural

$$\left.\begin{array}{l}[\![\_]\!]_{\mathcal{T}} \;:\; \mathit{Exp}\to\mathit{Type}\\ [\![\_]\!]_{\mathcal{T}} \;:\; \mathit{Set}[\mathit{ConfigItem}]\to\mathit{Type}\\ k_{\mathcal{T}}(\_) \;:\; K\to\mathit{Set}[\mathit{ConfigItem}]\\ \mathit{eqns}(\_) \;:\; \mathit{Set}[\mathit{Equation}]\to\mathit{Set}[\mathit{ConfigItem}]\\ \mathit{nextType}(\_) \;:\; \mathit{TypeVar}\to\mathit{Set}[\mathit{ConfigItem}]\\ t(\_) \;:\; \mathit{Nat}\to\mathit{TypeVar}\end{array}\right\}\cdots\left\{\begin{array}{lr}[\![\frac{E}{k_{\mathcal{T}}(E\curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))}]\!]_{\mathcal{T}} & (68)\\ [\![\langle k_{\mathcal{T}}(\tau)\ \mathit{eqns}(\cdot)\rangle]\!]_{\mathcal{T}} \equiv \tau & (69)\\ KK' \equiv K \curvearrowright \Box K' & (70)\\ KK' \equiv K' \curvearrowright K\Box & (71)\\ \tau\to K \equiv K \curvearrowright \tau\to\Box & (72)\end{array}\right.$$

11.2.2 Inference Rules

$$\left.\begin{array}{l}\underline{\lambda\_.\_} \;:\; \mathit{Var}\times\mathit{Exp}\to\mathit{Exp}\\ \underline{\_\ \_} \;:\; \mathit{Exp}\times\mathit{Exp}\to\mathit{Exp}\\ \underline{\mathsf{let}\ \_\ \mathsf{be}\ \_\ \mathsf{in}\ \_} \;:\; \mathit{Var}\times\mathit{Exp}\times\mathit{Exp}\to\mathit{Exp}\\ \mathit{TypeOf} \;:\; \mathit{Const}\to\mathit{TypeScheme}\end{array}\right\}\cdot\left\{\begin{array}{lr}k_{\mathcal{T}}(\frac{\lambda X\,.\,E}{\tau_v\to E[\tau_v/X]}\rangle\ \mathit{nextType}(\frac{\tau_v}{\mathit{next}(\tau_v)}) & (73)\\ k_{\mathcal{T}}(\frac{\tau\ \tau'}{\tau_v}\rangle\ \mathit{eqns}(\frac{\cdot}{\tau=\tau'\to\tau_v}\rangle\ \mathit{nextType}(\frac{\tau_v}{\mathit{next}(\tau_v)}) & (74)\\ k_{\mathcal{T}}(\frac{\mathsf{let}\,X\ \mathsf{be}\ E\ \mathsf{in}\ E'}{E'[E/X]}\rangle & (75)\\ k_{\mathcal{T}}(\frac{C}{\mathit{instantiate}(\mathit{TypeOf}(C))}\rangle & (76)\end{array}\right.$$

$$\left.\mathit{instantiate} \;:\; \mathit{TypeScheme}\to K\ \right\}\cdots\left\{\begin{array}{lr}k_{\mathcal{T}}(\frac{\mathit{instantiate}(\forall(\cdot).\tau)}{\tau}\rangle & (77)\\ k_{\mathcal{T}}(\mathit{instantiate}(\forall\frac{\langle\tau_v\rangle}{\cdot}.\frac{\tau}{\tau\,[\tau_v{}'/\tau_v]}))\rangle\ \mathit{nextType}(\frac{\tau_v{}'}{\mathit{next}(\tau_v{}')}) & (78)\end{array}\right.$$

$$\underline{\mathsf{Y}} \;:\; \to \mathit{Val} \;\}\left\{\; k_{\mathcal{T}}(\frac{\mathsf{Y}}{((\tau_v \to \mathit{next}(\tau_v)) \to \tau_v \to \mathit{next}(\tau_v)) \to \tau_v \to \mathit{next}(\tau_v)}\rangle\ \mathit{nextType}(\frac{\tau_v}{\mathit{next}(\mathit{next}(\tau_v))}) \right. \quad (79)$$

11.2.3 Equations

$$\_ = \_ \;:\; \mathit{Type} \times \mathit{Type} \to \mathit{Equation}\ [\mathit{comm}] \;\}\ldots\ldots \left\{ \begin{array}{lr} (\tau = \tau) = \cdot & (80) \\ \dfrac{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'}{(\tau_1 = \tau_1'), (\tau_2 = \tau_2')} & (81) \end{array} \right.$$

11.2.4 Unification

$$\mathit{solve} \;:\; \to K \;\}\ldots\ldots \left\{ \begin{array}{lr} k_{\mathcal{T}}(\dfrac{\tau_e}{\tau_e[\tau/\tau_v]} \curvearrowright \mathit{solve}\rangle\ \mathit{eqns}(\dfrac{(\tau_v = \tau)\mathcal{E}}{\mathcal{E}[\tau/\tau_v]}) \Leftarrow \tau_v \notin \mathit{vars}(\tau) & (82) \\ k_{\mathcal{T}}(\tau \curvearrowright \underset{\cdot}{\underline{\mathit{solve}}}\rangle\ \mathit{eqns}(\cdot) & (83) \end{array} \right.$$

11.3 Poly λ Calculus α

$$\alpha \;:\; \mathit{Result} \to \mathit{Result} \;\}\ldots\ldots \left\{ \begin{array}{lr} \alpha([\![E]\!]_{\mathcal{L}}) = [\![E]\!]_{\mathcal{T}} & (84) \\ \alpha([\![k_{\mathcal{L}}(K)]\!]_{\mathcal{L}}) = [\![k_{\mathcal{T}}(K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}} & (85) \\ \alpha(V : \mathit{Val}) = [\![V]\!]_{\mathcal{T}} & (86) \end{array} \right.$$

11.4 Poly λ Calculus Proofs

11.4.1 Preservation

Lemma 17 (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 84 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 59 $R = [\![E]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![E]\!]_{\mathcal{T}}$	84
$[\![k_{\mathcal{T}}(E \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	68

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

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(E \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	85

Rule 64 $R = [\![k_{\mathcal{L}}((\lambda X\,.\,E)(V : \mathit{Val}) \curvearrowright K)]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}((\lambda X\,.\,E)(V:\mathit{Val}) \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	85
$[\![k_{\mathcal{T}}(\lambda X\,.\,E \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	70
$[\![k_{\mathcal{T}}(t(0) \to E[t(0)/X] \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(1))]\!]_{\mathcal{T}}$	73
$[\![k_{\mathcal{T}}(E[t(0)/X] \curvearrowright t(0) \to \Box \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(1))]\!]_{\mathcal{T}}$	72
$[\![k_{\mathcal{T}}(\alpha \curvearrowright t(0) \to \Box \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(t(1))]\!]_{\mathcal{T}}$	assm.
$[\![k_{\mathcal{T}}(t(0) \to \alpha \curvearrowright \Box V \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(t(1))]\!]_{\mathcal{T}}$	72
$[\![k_{\mathcal{T}}((t(0) \to \alpha)\beta \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(t(1))]\!]_{\mathcal{T}}$	assm.
$[\![k_{\mathcal{T}}(t(1) \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}((t(0) \to \alpha) = (\beta \to t(1)) \cdot \mathcal{E})\ \mathit{nextType}(t(2))]\!]_{\mathcal{T}}$	74

We see $t(1) = \alpha$ by unification, and that $V \xrightarrow{*} t(0)$ and $E[t(0)/X] \xrightarrow{*} \alpha$

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

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(E[V/X] \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	85

By a lemma similar to the one we used in the simply-typed lambda calculus, we know $E[V/X] \xrightarrow{*} \alpha$.

Rule 65 $R = [\![k_{\mathcal{L}}(\mathsf{let}\ X\ \mathsf{be}\ V : \mathit{Val}\ \mathsf{in}\ E \curvearrowright K)]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}(\mathsf{let}\ X\ \mathsf{be}\ V : \mathit{Val}\ \mathsf{in}\ E \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	85
$[\![k_{\mathcal{T}}(E[V/X] \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	75

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

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(E[V/X] \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	85

Rule 66 $R = [\![k_{\mathcal{L}}((C : \mathit{Const})(V : \mathit{Val}) \curvearrowright K)]\!]_{\mathcal{L}}$

$\alpha(R)$	By Rule
$[\![k_{\mathcal{T}}((C : \mathit{Const})(V : \mathit{Val}) \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	85
$[\![k_{\mathcal{T}}((\mathit{instantiate}(\mathit{TypeOf}(C)))V \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	76
$[\![k_{\mathcal{T}}((\tau_1 \to \tau_2)V \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	assm.
$[\![k_{\mathcal{T}}((\tau_1 \to \tau_2)\tau' \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	assm.
$[\![k_{\mathcal{T}}(t(0) \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}((\tau_1 \to \tau_2 = \tau' \to t(0)) \cdot \mathcal{E})\ \mathit{nextType}(t(1))]\!]_{\mathcal{T}}$	74

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

$\alpha(R')$	By Rule
$[\![k_{\mathcal{T}}(\delta(C, V) \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	85
$[\![k_{\mathcal{T}}(\tau_2 \curvearrowright K \curvearrowright \mathit{solve})\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathcal{T}}$	above & δ-typeability

□

11.4.2 Progress

Lemma 18 (Progress)**.** *Let E be an expression such that $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some R. Either $R = V : \mathit{Val}$ for some V, or $\exists R'$ such that $R \longrightarrow R'$.*

Proof. Let E be an expression such that $[\![E]\!]_{\mathcal{T}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathcal{L}} \xrightarrow{*} R$ for some τ and R. We know by lemma 11 that any such R is equivalent under the structural rules to something of the form $[\![E']\!]_{\mathcal{L}}$ and therefore $[\![k_{\mathcal{L}}(E')]\!]_{\mathcal{L}}$ for some expression E'. This expression can be one of a few forms, namely, a variable, a lambda expression, or an application expression. We consider each individually.

(**X : *Var***) If E' is just a variable name, we know by preservation that this should type to τ, but there is no rule allowing us to type variables. Therefore, this case cannot arise.

($\lambda \boldsymbol{X} : \tau' \,.\, E''$) In this case, E' is a value so the property holds.

$(E'')(E''')$ We know by preservation that this should type to τ. The only way that is true is if E'' is a lambda term or if some subexpression of E'' is an application with a lambda expression on the left. In either case, we know we can take the corresponding step in the language.

We see that in all cases the property holds. $\square$

12 W

12.1 Exp Language

12.1.1 Structural Rules

$$\left.\begin{array}{l}
[\![\_]\!]_{\mathscr{E}} \;:\; \mathit{Exp} \to \mathit{Val} \\
[\![\_]\!]_{\mathscr{E}} \;:\; \mathit{Set}[\mathit{ConfigItem}] \to \mathit{Val} \\
k_{\mathscr{E}}(\_) \;:\; K \to \mathit{ConfigItem} \\
\underline{\_\ \_} \;:\; \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \\
\underline{\mathsf{let}\ \_\ \mathsf{be}\ \_\ \mathsf{in}\ \_} \;:\; \mathit{Var} \times \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \\
\underline{\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_} \;:\; \mathit{Exp} \times \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \\
\underline{\_ + \_} \;:\; \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}
\end{array}\right\}
\left\{\begin{array}{lr}
[\![E]\!]_{\mathscr{E}} \equiv [\![k_{\mathscr{E}}(E)]\!]_{\mathscr{E}} & (87) \\
[\![k_{\mathscr{E}}(V)]\!]_{\mathscr{E}} \equiv V & (88) \\
EE' \equiv E \curvearrowright \square E' & (89) \\
EE' \equiv E' \curvearrowright E\square & (90) \\
\mathsf{let}\ X\ \mathsf{be}\ E\ \mathsf{in}\ E' \equiv E \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \square\ \mathsf{in}\ E' & (91) \\
\mathsf{if}\ E\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 \equiv E \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 & (92) \\
K + K' \equiv K \curvearrowright \square + K' & (93) \\
K + K' \equiv K' \curvearrowright K + \square & (94)
\end{array}\right.$$

12.1.2 Semantic Rules

$$\underline{\mathsf{fix}\ \_\ .\ \_} \;:\; \mathit{Var} \times \mathit{Exp} \to \mathit{Exp} \;\Big\}\cdots\cdots
\left\{\begin{array}{lr}
k_{\mathscr{E}}(\dfrac{I : \mathit{Int} + I' : \mathit{Int}}{I +_{\mathit{int}} I'}\rangle & (95) \\
k_{\mathscr{E}}(\dfrac{(\lambda X\,.\,E)(V : \mathit{Val})}{E[V/X]}\rangle & (96) \\
k_{\mathscr{E}}(\dfrac{\mathsf{let}\ X\ \mathsf{be}\ V : \mathit{Val}\ \mathsf{in}\ E}{E[V/X]}\rangle & (97) \\
k_{\mathscr{E}}(\dfrac{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ E\ \mathsf{else}\ E'}{E}\rangle & (98) \\
k_{\mathscr{E}}(\dfrac{\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ E\ \mathsf{else}\ E'}{E'}\rangle & (99) \\
k_{\mathscr{E}}(\dfrac{\mathsf{fix}\ X\,.\,E}{E[\mathsf{fix}\ X\,.\,E/X]}\rangle & (100)
\end{array}\right.$$

12.2 W

12.2.1 Structural Rules

$$\left.\begin{array}{l}
[\![\_]\!]_{\mathscr{W}} \;:\; \mathit{Exp} \to \mathit{Type} \\
[\![\_]\!]_{\mathscr{W}} \;:\; \mathit{Set}[\mathit{ConfigItem}] \to \mathit{Type} \\
k_{\mathscr{W}}(\_) \;:\; K \to \mathit{ConfigItem} \\
\mathit{eqns}(\_) \;:\; \mathit{Set}[\mathit{Equation}] \to \mathit{ConfigItem} \\
\mathit{nextType}(\_) \;:\; \mathit{TypeVar} \to \mathit{ConfigItem} \\
t(\_) \;:\; \mathit{Nat} \to \mathit{TypeVar}
\end{array}\right\}
\left\{\begin{array}{lr}
[\![E]\!]_{\mathscr{W}} \equiv [\![k_{\mathscr{W}}(E)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}} & (101) \\
[\![\langle k_{\mathscr{W}}(\tau)\ \mathit{eqns}(\mathcal{E})\rangle]\!]_{\mathscr{W}} \equiv \mathcal{E}[\tau] & (102) \\
EE' \equiv E \curvearrowright \square E' & (103) \\
EE' \equiv E' \curvearrowright E\square & (104) \\
\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' \equiv K \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ S\ \mathsf{else}\ S' & (105) \\
\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' \equiv S \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ \square\ \mathsf{else}\ S' & (106) \\
\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' \equiv S' \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ \square & (107) \\
\mathsf{let}\ X\ \mathsf{be}\ K\ \mathsf{in}\ K' \equiv K \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \square\ \mathsf{in}\ K' & (108) \\
\tau \to K \equiv K \curvearrowright \tau \to \square & (109)
\end{array}\right.$$

12.2.2 Inference Rules

$$\left.\begin{array}{l} \mathit{restore}(\_)\ :\ \mathit{Environment} \to K \\ \textsc{Int}\ :\ \ \to \mathit{Type} \\ \textsc{Bool}\ :\ \ \to \mathit{Type} \end{array}\right\} \cdots\cdots\cdots \left\{\begin{array}{rr} k_{\mathscr{W}}(\tau \curvearrowright \underset{\cdot}{\underline{\mathit{restore}(\Gamma)}}\rangle\ \mathit{env}(\underset{\Gamma}{\underline{\_}}) & (110) \\ I : \mathit{Int} \longrightarrow \textsc{Int} & (111) \\ \mathsf{true} \longrightarrow \textsc{Bool} & (112) \\ \mathsf{false} \longrightarrow \textsc{Bool} & (113) \end{array}\right.$$

$$\}\left\{\begin{array}{r} k_{\mathscr{W}}(\underset{(\mathcal{E}[\tau])[tl \leftarrow tl']}{\underline{\quad X \quad}}\rangle\ \mathit{tenv}(\Gamma)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\underset{\tau_v + |tl|}{\underline{\quad\tau_v\quad}}) \Longleftarrow \Gamma[X] = \mathsf{let}(t), tl = \mathit{vars}(\mathcal{E}[\tau]) - \mathit{vars}(\Gamma) \text{ and} \\ tl' = \tau_v \ldots (\tau_v + |tl| - 1)\ (114) \end{array}\right.$$

$$\left.\begin{array}{l} \underline{\lambda\_ . \_}\ :\ \mathit{Var} \times \mathit{Exp} \to \mathit{Exp} \\ \_ \to \_\ :\ \mathit{Type} \times \mathit{Type} \to \mathit{Type} \end{array}\right\}\left\{ k_{\mathscr{W}}(\underset{(\tau_v \to E) \curvearrowright \mathit{restore}(\Gamma)}{\underline{\quad \lambda X . E \quad}}\rangle\ \mathit{tenv}(\underset{\Gamma[X \leftarrow \tau_v]}{\underline{\quad\Gamma\quad}})\ \mathit{nextType}(\underset{\mathit{next}(\tau_v)}{\underline{\quad\tau_v\quad}})\ (115) \right.$$

$$\left.\begin{array}{l} \underline{\_\,\_}\ :\ \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \\ \underline{\mathsf{let}\ \_\ \mathsf{be}\ \_\ \mathsf{in}\ \_}\ :\ \mathit{Var} \times \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp} \end{array}\right\}\left\{\begin{array}{rr} k_{\mathscr{W}}(\underset{\textsc{Int}}{\underline{\tau + \tau'}}\rangle\ \mathit{eqns}\langle\underset{\tau = \textsc{Int} \cdot \tau' = \textsc{Int}}{\underline{\quad\cdot\quad}}\rangle & (116) \\ k_{\mathscr{W}}(\underset{\tau_v}{\underline{\tau_1\tau_2}}\rangle\ \mathit{eqns}(\underset{\tau_1 = \tau_2 \to \tau_v}{\underline{\quad\cdot\quad}})\ \mathit{nextType}(\underset{\mathit{next}(\tau_v)}{\underline{\quad\tau_v\quad}}) & (117) \\ k_{\mathscr{W}}(\underset{E \curvearrowright \mathit{restore}(\Gamma)}{\underline{\mathsf{let}\,X\ \mathsf{be}\ \tau\ \mathsf{in}\ E}}\rangle\ \mathit{tenv}(\underset{\Gamma[X \leftarrow \mathit{let}(\tau_v)]}{\underline{\quad\Gamma\quad}}) & (118) \\ k_{\mathscr{W}}(\underset{\tau_1}{\underline{\mathsf{if}\ \tau\ \mathsf{then}\ \tau_1\ \mathsf{else}\ \tau_2}}\rangle\ \mathit{eqns}\langle\underset{\tau = \textsc{Bool} \cdot \tau_1 = \tau_2}{\underline{\quad\cdot\quad}}\rangle & (119) \end{array}\right.$$

12.2.3 Abstract Structural Rules

$$[\![E]\!]_{\widehat{\mathscr{W}}} \equiv [\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\cdot)\ \mathit{subst}(\mathit{ID})]\!]_{\widehat{\mathscr{W}}}\ (120)$$

$$[\![\langle k_{\widehat{\mathscr{W}}}(\tau)\ \mathit{subst}(\theta)\rangle]\!]_{\widehat{\mathscr{W}}} \equiv \theta(\tau)\ (121)$$

$$EE' \equiv E \curvearrowright \square E'\ (122)$$

$$EE' \equiv E' \curvearrowright E\square\ (123)$$

$$\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' \equiv K \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ S\ \mathsf{else}\ S'\ (124)$$

$$\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' \equiv S \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ \square\ \mathsf{else}\ S'\ (125)$$

$$\mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ S' \equiv S' \curvearrowright \mathsf{if}\ K\ \mathsf{then}\ S\ \mathsf{else}\ \square\ (126)$$

$$\mathsf{let}\,X\ \mathsf{be}\ K\ \mathsf{in}\ K' \equiv K \curvearrowright \mathsf{let}\,X\ \mathsf{be}\ \square\ \mathsf{in}\ K'\ (127)$$

$$\tau \to K \equiv K \curvearrowright \tau \to \square\ (128)$$

12.2.4 Abstract Inference Rules

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

$$\theta \oplus (\tau_1 = \tau_2) = \theta \otimes (\theta(\tau_1) = \theta(\tau_2))$$

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

We define the reduction relation $\overset{\wedge}{\longrightarrow}$ as $\overset{\wedge}{\longrightarrow} = \longrightarrow ; \theta$. We further identify configurations $[\![k_{\widehat{\mathscr{W}}}(K_1)\ \mathit{tenv}(\Gamma_1)\ \mathit{subst}(\theta_1)]\!]_{\widehat{\mathscr{W}}}$ and $[\![k_{\widehat{\mathscr{W}}}(K_2)\ \mathit{tenv}(\Gamma_2)\ \mathit{subst}(\theta_2)]\!]_{\widehat{\mathscr{W}}}$ 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 $\iota(\theta_1) = \theta_2$.

$$k_{\widehat{\mathscr{W}}}(\tau \curvearrowright \frac{\mathit{restore}(\Gamma)}{\cdot}\rangle\ \mathit{tenv}(\frac{\_}{\Gamma}) \tag{129}$$

$$I : \mathit{Int} \longrightarrow \textsc{Int} \tag{130}$$

$$\mathsf{true} \longrightarrow \textsc{Bool} \tag{131}$$

$$\mathsf{false} \longrightarrow \textsc{Bool} \tag{132}$$

$$k_{\widehat{\mathscr{W}}}(\frac{X}{\tau[\mathit{tl} \leftarrow \mathit{tl}']}\rangle\ \mathit{tenv}(\Gamma) \text{ where } \Gamma[X] = \mathsf{let}(\tau),\ \mathit{tl} = \mathit{vars}(\tau) - \mathit{vars}(\Gamma), \text{ and } \mathit{tl}' = |\mathit{tl}| \text{ fresh type variables} \tag{133}$$

$$k_{\widehat{\mathscr{W}}}(\frac{X}{\Gamma[X]}\rangle\ \mathit{tenv}(\Gamma) \text{ where } \Gamma[X] \neq \mathsf{let}(\_) \tag{134}$$

$$k_{\widehat{\mathscr{W}}}(\frac{\lambda X\,.\,E}{(\tau_v \to E) \curvearrowright \mathit{restore}(\Gamma)}\rangle\ \mathit{tenv}(\frac{\Gamma}{\Gamma[X \leftarrow \tau_v]}) \text{ where } \tau_v \text{ is a fresh type variable} \tag{135}$$

$$k_{\widehat{\mathscr{W}}}(\frac{\mathsf{let}\,X \text{ } \mathsf{be}\ \tau\ \mathsf{in}\ E}{E \curvearrowright \mathit{restore}(\Gamma)}\rangle\ \mathit{tenv}(\frac{\Gamma}{\Gamma[X \leftarrow \mathit{let}(\tau_v)]}) \text{ where } \tau_v \text{ is a fresh type variable} \tag{136}$$

$$k_{\widehat{\mathscr{W}}}(\frac{\mathsf{fix}\ X\,.\,E}{E \curvearrowright ?_{=}(\tau_v) \curvearrowright \mathit{restore}(\Gamma)}\rangle\ \mathit{tenv}(\frac{\Gamma}{\Gamma[X \leftarrow \tau_v]}) \text{ where } \tau_v \text{ is a fresh type variable} \tag{137}$$

$$k_{\widehat{\mathscr{W}}}(\tau \curvearrowright \frac{?_{=}(\tau_v)}{\cdot}\rangle\ \mathit{subst}(\frac{\theta}{\theta \oplus (\tau = \tau_v)}) \tag{138}$$

$$k_{\widehat{\mathscr{W}}}(\frac{\tau + \tau'}{\textsc{Int}}\rangle\ \mathit{subst}(\frac{\theta}{\theta \oplus (\tau = \textsc{Int}) \oplus (\tau' = \textsc{Int})}) \tag{139}$$

$$k_{\widehat{\mathscr{W}}}(\frac{\tau_1 \tau_2}{\tau_v}\rangle\ \mathit{subst}(\frac{\theta}{\theta \oplus (\tau_1 = \tau_2 \to \tau_v)}) \text{ where } \tau_v \text{ is a fresh type variable} \tag{140}$$

$$k_{\widehat{\mathscr{W}}}(\frac{\mathsf{if}\ \tau\ \mathsf{then}\ \tau_1\ \mathsf{else}\ \tau_2}{\tau_1}\rangle\ \mathit{subst}(\frac{\theta}{\theta'}) \text{ where } \theta' = \theta \oplus (\tau_1 = \tau_2) \oplus (\tau = \textsc{Bool}) \tag{141}$$

12.3 $\alpha : \boldsymbol{Set}[\boldsymbol{ConfigItem}]_{\mathscr{E}} \to \boldsymbol{Set}[\boldsymbol{ConfigItem}]_{\mathscr{W}}$

A distinguishing feature of our technique is that we use an abstraction function, α, to enable us to convert between a configuration in the language domain to a corresponding configuration in the typing domain. Using an abstraction function in proving soundness is a technique used frequently in the domain of processor construction, as introduced in [HSG98], or compiler optimization [KSK06, KSK07].

Lemma 19. *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. □

We now define α:

$$\alpha([\![E]\!]_{\mathscr{E}}) = [\![E]\!]_{\mathscr{W}} \tag{142}$$

By lemma 19, we know this definition of α is well-defined for all reachable configurations, and homomorphic with respect to structural rules.

12.4 Proofs

12.4.1 Preservation

Definition 1 (Generalizes relation)**.** A type τ is said to *generalize* a type τ' (written $\tau \succ \tau'$) if $\exists\theta.\theta(\tau) = \tau'$ 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 20.
$$[\![k_{\widehat{\mathscr{W}}}(K_1)\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \hat{\longrightarrow} [\![k_{\widehat{\mathscr{W}}}(K_2)\ \mathit{tenv}(\Gamma')\ \mathit{subst}(\theta')]\!]_{\widehat{\mathscr{W}}}$$
iff
$$[\![k_{\widehat{\mathscr{W}}}(K_1 \curvearrowright K)\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \hat{\longrightarrow} [\![k_{\widehat{\mathscr{W}}}(K_2 \curvearrowright \theta'(K))\ \mathit{tenv}(\Gamma')\ \mathit{subst}(\theta')]\!]_{\widehat{\mathscr{W}}}$$

Lemma 21. $\theta \succ \theta \oplus \mathcal{E}$

Lemma 22. *If*
$$[\![k_{\widehat{\mathscr{W}}}(K)\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \hat{\longrightarrow} [\![k_{\widehat{\mathscr{W}}}(K')\ \mathit{tenv}(\Gamma')\ \mathit{subst}(\theta')]\!]_{\widehat{\mathscr{W}}}$$
then $\theta \succ \theta'$.

Lemma 23. *If*
$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau)\ \mathit{tenv}(\Gamma')\ \mathit{subst}(\theta')]\!]_{\widehat{\mathscr{W}}}$$
then $\Gamma' = \theta'(\Gamma)$.

Lemma 24. *If*
$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_1)\ \mathit{tenv}(\Gamma')\ \mathit{subst}(\theta')]\!]_{\widehat{\mathscr{W}}}$$
then if
$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma[X \leftarrow \tau])\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_2)\ \mathit{tenv}(\Gamma'')\ \mathit{subst}(\theta'')]\!]_{\widehat{\mathscr{W}}}$$
for some fresh type variable τ, *we have that* $\tau_2 \succ \tau_1$.

Lemma 25. *If*
$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma[X \leftarrow \tau])\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_1)\ \mathit{tenv}(\Gamma')\ \mathit{subst}(\theta')]\!]_{\widehat{\mathscr{W}}}$$
then if
$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma[X \leftarrow \tau'])\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_2)\ \mathit{tenv}(\Gamma'')\ \mathit{subst}(\theta'')]\!]_{\widehat{\mathscr{W}}}$$
for types $\tau' \succ \tau$, *we have that* $\tau_2 \succ \tau_1$.

Lemma 26. *If*
$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_1)\ \mathit{tenv}(\theta'(\Gamma))\ \mathit{subst}(\theta')]\!]_{\widehat{\mathscr{W}}}$$
and E *contains no* X, *then*
$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma[X \leftarrow \tau])\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_1)\ \mathit{tenv}(\theta'(\Gamma)[X \leftarrow \tau])\ \mathit{subst}(\theta')]\!]_{\widehat{\mathscr{W}}}$$
for a fresh τ.

Lemma 27. *If*
$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_1)\ \mathit{tenv}(\Gamma_1)\ \mathit{subst}(\theta_1)]\!]_{\widehat{\mathscr{W}}}$$
and τ *is a fresh type variable, then for any* X,
$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma[X \leftarrow \tau])\ \mathit{subst}(\theta)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_2)\ \mathit{tenv}(\Gamma_2)\ \mathit{subst}(\theta_2)]\!]_{\widehat{\mathscr{W}}}$$
and $\tau_2 \succ \tau_1$ *and* $\theta_2 \succ \theta_1$.

Lemma 28 (Env $\Rightarrow$ Rep). *If*

- $[\![k_{\widehat{\mathscr{W}}}(V)\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta_0)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_V)\ \mathit{tenv}(\Gamma_1)\ \mathit{subst}(\theta_1)]\!]_{\widehat{\mathscr{W}}}$
- $[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma_1[X \leftarrow \tau_V])\ \mathit{subst}(\theta_1)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_E)\ \mathit{tenv}(\Gamma_2)\ \mathit{subst}(\theta_2)]\!]_{\widehat{\mathscr{W}}}$

then $[\![k_{\widehat{\mathscr{W}}}(E[X \leftarrow V])\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta_0)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_R)\ \mathit{tenv}(\Gamma_3)\ \mathit{subst}(\theta_3)]\!]_{\widehat{\mathscr{W}}})$ *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

$$[\![k_{\widehat{\mathscr{W}}}(V)\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta_0)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_V)\ \mathit{tenv}(\Gamma_1)\ \mathit{subst}(\theta_1)]\!]_{\widehat{\mathscr{W}}}) \tag{143}$$

and

$$[\![k_{\widehat{\mathscr{W}}}(\lambda Y\,.\,E)\ \mathit{tenv}(\Gamma_1[X \leftarrow \tau_V])\ \mathit{subst}(\theta_1)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(E \curvearrowright K)\ \mathit{tenv}(\Gamma_1[X \leftarrow \tau_V][Y \leftarrow \tau])\ \mathit{subst}(\theta_1)]\!]_{\widehat{\mathscr{W}}} \tag{144}$$

$$\xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_E \curvearrowright \theta_2(K))\ \mathit{tenv}(\Gamma')\ \mathit{subst}(\theta_2)]\!]_{\widehat{\mathscr{W}}} \tag{145}$$

$$\xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\theta_2(\tau) \to \tau_E)\ \mathit{tenv}(\Gamma'')\ \mathit{subst}(\theta_2)]\!]_{\widehat{\mathscr{W}}} \tag{146}$$

$$\tag{147}$$

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

$$[\![k_{\widehat{\mathscr{W}}}((\lambda Y\,.\,E)[X \leftarrow V])\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta_0)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_R)\ \mathit{tenv}(\Gamma_3)\ \mathit{subst}(\theta_3)]\!]_{\widehat{\mathscr{W}}} \tag{148}$$

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 26 and assumption 143, we know that

$$[\![k_{\widehat{\mathscr{W}}}(V)\ \mathit{tenv}(\Gamma[Y \leftarrow \tau])\ \mathit{subst}(\theta_0)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_V)\ \mathit{tenv}(\Gamma_1[Y \leftarrow \tau])\ \mathit{subst}(\theta_1)]\!]_{\widehat{\mathscr{W}}}) \tag{149}$$

Furthermore, by lemma 27,

$$[\![k_{\widehat{\mathscr{W}}}(E)\ \mathit{tenv}(\Gamma_1[Y \leftarrow \tau][X \leftarrow \tau_V])\ \mathit{subst}(\theta_1)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau'_E)\ \mathit{tenv}(\Gamma'_2)\ \mathit{subst}(\theta'_2)]\!]_{\widehat{\mathscr{W}}} \tag{150}$$

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

$$[\![k_{\widehat{\mathscr{W}}}(E[X \leftarrow V])\ \mathit{tenv}(\Gamma[Y \leftarrow \tau])\ \mathit{subst}(\theta_0)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\tau_S)\ \mathit{tenv}(\Gamma_3)\ \mathit{subst}(\theta_3)]\!]_{\widehat{\mathscr{W}}} \tag{151}$$

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

So now we know that

$$[\![k_{\widehat{\mathscr{W}}}((\lambda Y\,.\,E)[X \leftarrow V])\ \mathit{tenv}(\Gamma)\ \mathit{subst}(\theta_0)]\!]_{\widehat{\mathscr{W}}} \xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(E[X \leftarrow V] \curvearrowright K)\ \mathit{tenv}(\Gamma[Y \leftarrow \tau])\ \mathit{subst}(\theta_0)]\!]_{\widehat{\mathscr{W}}} \tag{152}$$

$$\xrightarrow{\hat{*}} [\![k_{\widehat{\mathscr{W}}}(\theta_3(\tau) \to \tau_S)\ \mathit{tenv}(\Gamma_3)\ \mathit{subst}(\theta_3)]\!]_{\widehat{\mathscr{W}}} \tag{153}$$

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 will construct an appropriate unifier θ''. Assume $\tau \in \tau_S$. Then $\theta_3(\tau) = \tau$ by 4.

$$\theta''(\tau_v) = \begin{cases} \theta_2(\tau_v) & \text{if } \tau_v = \tau \\ \theta'(\tau_v) & \text{if } \tau_v \in vars(\tau_S) \end{cases} \tag{154}$$

This is well-defined if $\theta_2(\tau) = \theta'(\tau)$.

Assume $\tau \in \tau_E$. We know $\theta_2(\tau) = \tau$ by 3 and $\theta'(\tau_E) = \tau_E$ by 2, and since $\tau \in \tau_E$, $\theta'(\tau) = \tau$.

Assume $\tau \notin \tau_E$. By 1 we know $\theta(\tau) = \theta_2(\tau)$.

□

Lemma 29. *If $\mathscr{W} \models \alpha(V) \xrightarrow{*} \tau$ then $[\![V]\!]_{\mathscr{W}} \xrightarrow{*} \tau$*

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

Lemma 30. *If $[\![E]\!]_{\mathscr{W}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathscr{E}} \xrightarrow{*} R$ for some τ and R, then $\mathscr{W} \models \alpha(R) \xrightarrow{*} \tau'$ for some τ' unifiable with τ.*

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

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

Induction Case Assume $[\![E]\!]_{\mathscr{E}} \xrightarrow{n} R$ 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 87 through 94 These all follow from lemma 19.

Rule 95 $R = [\![k_{\mathscr{E}}(I : Int + I' : Int \curvearrowright K)]\!]_{\mathscr{E}}$ Now we work with $\alpha(R)$:

$$\alpha(R) = \alpha([\![k_{\mathscr{E}}(I + I' \curvearrowright K)]\!]_{\mathscr{E}})$$

which reduces to:

$$[\![k_{\mathscr{W}}(I + I' \curvearrowright K)\ tenv(\cdot)\ eqns(\cdot)\ nextType(0)]\!]_{\mathscr{W}}$$

by the definition of α. This then reduces to:

$$[\![k_{\mathscr{W}}(\textsc{Int} + \textsc{Int} \curvearrowright K)\ tenv(\cdot)\ eqns(\cdot)\ nextType(0)]\!]_{\mathscr{W}}$$

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

$$[\![k_{\mathscr{W}}(\textsc{Int} \curvearrowright K)\ tenv(\cdot)\ eqns(\textsc{Int} = \textsc{Int}, \textsc{Int} = \textsc{Int})\ nextType(0)]\!]_{\mathscr{W}}$$

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

$$[\![k_{\mathscr{W}}(\textsc{Int} \curvearrowright K)\ tenv(\cdot)\ eqns(\cdot)\ nextType(0)]\!]_{\mathscr{W}}$$

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

$$\alpha(R') = \alpha([\![k_{\mathscr{E}}(I +_{int} I' \curvearrowright K)]\!]_{\mathscr{E}})$$

which reduces to:

$$[\![k_{\mathscr{W}}(I +_{int} I' \curvearrowright K)\ tenv(\cdot)\ eqns(\cdot)\ nextType(0)]\!]_{\mathscr{W}}$$

by the definition of α. This immediately reduces to:

$$[\![k_{\mathscr{W}}(\textsc{Int} \curvearrowright K)\ tenv(\cdot)\ eqns(\cdot)\ nextType(0)]\!]_{\mathscr{W}}$$

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 96 $R = [\![k_{\mathscr{E}}((\lambda X\,.\,E)(V : \mathit{Val}) \curvearrowright K)]\!]_{\mathscr{E}}$

$\alpha(R)$	By Rule
$[\![k_{\mathscr{W}}((\lambda X\,.\,E)(V : \mathit{Val}) \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	142
$[\![k_{\mathscr{W}}(\lambda X\,.\,E \curvearrowright \square V \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	122
$[\![k_{\mathscr{W}}((t(0) \to E) \curvearrowright \mathit{restore}(\cdot) \curvearrowright \square V \curvearrowright K)\ \mathit{tenv}([X, t(0)])\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(1))]\!]_{\mathscr{W}}$	135
$[\![k_{\mathscr{W}}(E \curvearrowright (t(0) \to \square) \curvearrowright \mathit{restore}(\cdot) \curvearrowright \square V \curvearrowright K)\ \mathit{tenv}([X, t(0)])\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(1))]\!]_{\mathscr{W}}$	128
$[\![k_{\mathscr{W}}(\tau_1 \curvearrowright (t(0) \to \square) \curvearrowright \mathit{restore}(\cdot) \curvearrowright \square V \curvearrowright K)\ \mathit{tenv}([X, t(0)])\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	ind assum
$[\![k_{\mathscr{W}}((t(0) \to \tau_1) \curvearrowright \mathit{restore}(\cdot) \curvearrowright \square V \curvearrowright K)\ \mathit{tenv}([X, t(0)])\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	128
$[\![k_{\mathscr{W}}((t(0) \to \tau_1) \curvearrowright \square V \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	129
$[\![k_{\mathscr{W}}((t(0) \to \tau_1)V \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	122
$[\![k_{\mathscr{W}}(V \curvearrowright (t(0) \to \tau_1)\square \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	123
$[\![k_{\mathscr{W}}(\tau_2 \curvearrowright (t(0) \to \tau_1)\square \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E}')\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$	ind assum
$[\![k_{\mathscr{W}}((t(0) \to \tau_1)\tau_2 \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E}')\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$	123
$[\![k_{\mathscr{W}}(\tau_v{}' \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E}' \cdot (t(0) \to \tau_1 = \tau_2 \to \tau_v{}'))\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$	140

In the above we see that

- $[\![k_{\mathscr{W}}(E \curvearrowright K_1)\ \mathit{tenv}([X, t(0)])\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(1))]\!]_{\mathscr{W}} \xrightarrow{*}$
  $[\![k_{\mathscr{W}}(\tau_1 \curvearrowright K_1)\ \mathit{tenv}([X, t(0)])\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$
- $[\![k_{\mathscr{W}}(V \curvearrowright K_2)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}} \xrightarrow{*}$
  $[\![k_{\mathscr{W}}(\tau_2 \curvearrowright K_2)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E}')\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$.
- $t(0) = \tau_2$
- $\tau_1 = \tau_v{}'$

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

$\alpha(R')$	By Rule
$[\![k_{\mathscr{W}}(E[V/X] \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	142
$[\![k_{\mathscr{W}}(\tau_v{}' \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	Lemma 28

Rule 97 $R = [\![k_{\mathscr{E}}(\mathsf{let}\ X\ \mathsf{be}\ V : \mathit{Val}\ \mathsf{in}\ E \curvearrowright K)]\!]_{\mathscr{E}}$

$\alpha(R)$	By Rule
$[\![k_{\mathscr{E}}(\mathsf{let}\ X\ \mathsf{be}\ V\ \mathsf{in}\ E \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	142
$[\![k_{\mathscr{W}}(V \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \square\ \mathsf{in}\ E \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	127
$[\![k_{\mathscr{W}}(\tau_1 \curvearrowright \mathsf{let}\ X\ \mathsf{be}\ \square\ \mathsf{in}\ E \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	ind assum & Lemma
$[\![k_{\mathscr{W}}(\mathsf{let}\ X\ \mathsf{be}\ \tau_1\ \mathsf{in}\ E \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	127
$[\![k_{\mathscr{W}}(E \curvearrowright \mathit{restore}(\cdot) \curvearrowright K)\ \mathit{tenv}([X, \mathit{let}(\tau_1)])\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	136
$[\![k_{\mathscr{W}}(\tau_2 \curvearrowright \mathit{restore}(\cdot) \curvearrowright K)\ \mathit{tenv}([X, \mathit{let}(\tau_1)])\ \mathit{eqns}(\mathcal{E}')\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$	ind assum
$[\![k_{\mathscr{W}}(\tau_2 \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E}')\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$	129

In the above we see that

- $[\![k_{\mathscr{W}}(E \curvearrowright K_1)\ \mathit{tenv}([X, \mathit{let}(\tau_1)])\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}} \xrightarrow{*}$
  $[\![k_{\mathscr{W}}(\tau_2 \curvearrowright K_1)\ \mathit{tenv}([X, \mathit{let}(\tau_1)])\ \mathit{eqns}(\mathcal{E}')\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$
- $[\![k_{\mathscr{W}}(V \curvearrowright K_2)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}} \xrightarrow{*}$
  $[\![k_{\mathscr{W}}(\tau_1 \curvearrowright K_2)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E}')\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$.

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

$\alpha(R')$	By Rule
$[\![k_{\mathscr{W}}(E[V/X] \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	142
$[\![k_{\mathscr{W}}(\tau_v{}' \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	Lemma 28

Rule 98 $R = [\![k_{\mathscr{E}}(\textsf{if true then } E \textsf{ else } E' \curvearrowright K)]\!]_{\mathscr{E}}$

$\alpha(R)$	By Rule
$[\![k_{\mathscr{E}}(\textsf{if true then } E \textsf{ else } E' \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	142
$[\![k_{\mathscr{E}}(\textsf{if } \textsc{Bool} \textsf{ then } E \textsf{ else } E' \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	131
$[\![k_{\mathscr{E}}(E \curvearrowright \textsf{if } \textsc{Bool} \textsf{ then } \square \textsf{ else } E' \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	125
$[\![k_{\mathscr{E}}(\tau \curvearrowright \textsf{if } \textsc{Bool} \textsf{ then } \square \textsf{ else } E' \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	ind assum
$[\![k_{\mathscr{E}}(\textsf{if } \textsc{Bool} \textsf{ then } \tau \textsf{ else } E' \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	125
$[\![k_{\mathscr{E}}(E' \curvearrowright \textsf{if } \textsc{Bool} \textsf{ then } \tau \textsf{ else } \square \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	126
$[\![k_{\mathscr{E}}(\tau' \curvearrowright \textsf{if } \textsc{Bool} \textsf{ then } \tau \textsf{ else } \square \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E}')\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$	ind assum
$[\![k_{\mathscr{E}}(\textsf{if } \textsc{Bool} \textsf{ then } \tau \textsf{ else } \tau' \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E}')\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$	126
$[\![k_{\mathscr{E}}(\tau \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E}' \cdot \textsc{Bool} = \textsc{Bool} \cdot \tau = \tau')\ \mathit{nextType}(\tau_v{}')]\!]_{\mathscr{W}}$	141

$R' = [\![k_{\mathscr{E}}(E \curvearrowright K)]\!]_{\mathscr{E}}$

$\alpha(R')$	By Rule
$[\![k_{\mathscr{W}}(E \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	142
$[\![k_{\mathscr{W}}(\tau \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\mathcal{E})\ \mathit{nextType}(\tau_v)]\!]_{\mathscr{W}}$	topOfStack for types

Rule 99 This proceeds like rule 98.

Rule 100 $R = [\![k_{\mathscr{E}}(\textsf{fix } X \,.\, E \curvearrowright K)]\!]_{\mathscr{E}}$

$\alpha(R)$	By Rule
$[\![k_{\mathscr{E}}(\textsf{fix } X \,.\, E \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	142

$R' = [\![k_{\mathscr{E}}(E[\textsf{fix } X \,.\, E/X] \curvearrowright K)]\!]_{\mathscr{E}}$

$\alpha(R')$	By Rule
$[\![k_{\mathscr{W}}(E[\textsf{fix } X \,.\, E/X] \curvearrowright K)\ \mathit{tenv}(\cdot)\ \mathit{eqns}(\cdot)\ \mathit{nextType}(t(0))]\!]_{\mathscr{W}}$	142

□

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

Proof. This follows directly from lemmas 29 and 30. □

12.4.2 Progress

Lemma 31. *If* $[\![k_{\mathscr{E}}(E \curvearrowright K_1)]\!]_{\mathscr{E}} \xrightarrow{*} [\![k_{\mathscr{E}}(V \curvearrowright K_1)]\!]_{\mathscr{E}}$*, then* $[\![k_{\mathscr{E}}(E \curvearrowright K_2)]\!]_{\mathscr{E}} \xrightarrow{*} [\![k_{\mathscr{E}}(V \curvearrowright K_2)]\!]_{\mathscr{E}}$ *for any* K_1 *and* K_2*.*

Proof. This follows by simple inspection of the reduction rules. □

Lemma 32. *If* $[\![V]\!]_{\mathscr{W}} \xrightarrow{*} \tau$ *and* τ *is unifiable with* Int*, then* $\tau = \textsc{Int}$*.*

Proof. The only things unifiable with Int are Int itself and an arbitrary τ. Because there is no way to type a value to an arbitrary τ, it must type to Int itself. □

Lemma 33. *If* $E_1 + E_2$ *is an expression such that* $[\![E_1 + E_2]\!]_{\mathscr{W}} \xrightarrow{*} \tau$*, then if* $[\![E_1]\!]_{\mathscr{E}} \xrightarrow{*} V$*, we have that* $[\![V]\!]_{\mathscr{W}} \xrightarrow{*} \textsc{Int}$*.*

Proof. This follows by first noting there is no way to reduce an addition expression on the top of the stack to a type except by applying the rule 139. Therefore, we know that $[\![E_1]\!]_{\mathscr{W}} \xrightarrow{*} \tau_1$. By preservation, we then know that if $[\![E_1]\!]_{\mathscr{E}} \xrightarrow{*} V$, it must be the case that $[\![V]\!]_{\mathscr{W}} \xrightarrow{*} \tau'$ for some τ' unifiable with τ. By lemma 32, we then know $\tau = \textsc{Int}$ and we are done. □

Definition 3 (Key Subexpression). Any subexpression of an expression that must be reduced to a value before the entire expression can be reduced is called a *key subexpression*. For example, E is the only key expression in if E then E_1 else E_2, but both E and E' are key subexpressions in EE'.

Lemma 34. *Let E be an expression such that $[\![E]\!]_{\mathscr{W}} \xrightarrow{*} \tau$. For any key subexpression S of E, if $[\![k_{\mathscr{E}}(S \curvearrowright K)]\!]_{\mathscr{E}} \xrightarrow{*} [\![k_{\mathscr{E}}(V \curvearrowright K)]\!]_{\mathscr{E}}$ for some K, then V has the expected type.*

Proof. This follows from lemma 31 and all lemmas like 33 for each key subexpression of each expression. □

Theorem 2 (Progress)**.** *For any expression E where $[\![E]\!]_{\mathscr{W}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathscr{E}} \xrightarrow{*} R$ for some τ and R, either $R = V$ for some V, or $\exists R'$ such that $R \longrightarrow R'$.*

Proof. Let E be an expression such that $[\![E]\!]_{\mathscr{W}} \xrightarrow{*} \tau$ and $[\![E]\!]_{\mathscr{E}} \xrightarrow{*} R$ for some τ and R.

The proof proceeds by induction on the size of the expression E for configuration R implied by lemma 19.

Base Case If there are no operators in E, then the expression is a value and we are done. If there is one operator in E, then the key subexpressions must all be values, and by lemma 34 and inspection of our rewrite rules to ensure we covered all expressions, we know we can apply the appropriate reduction.

Induction Case Assume all expressions of size less than n that type are either values or can take a step. Consider an expression of size n that types. If all of its key subexpressions are values, then by lemma 34 we can apply one of the language rules on the expression directly. If not, then there must be a key subexpression that is not a value, so we can move it in front using a structural rule, then by inductive assumption, we know we can take a step on that expression.

□

References

[ABF$^+$05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *TPHOLs '05*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005.

[AMGR93] Andrew Appel, David MacQueen, Lal George, and John Reppy. Standard ML of New Jersey release notes (version 0.93). Technical report, AT&T Bell Laboratories, November 1993.

[Ano07a] Anonymous. -1-. Technical Report —, —, 2007.

[Ano07b] Anonymous. -2-. Technical Report —, —, 2007.

[Ban92] Richard Banach. Simple type inference for term graph rewriting systems. In *CTRS '92*, volume 656 of *LNCS*, pages 51–66, 1992.

[Bar91] Henk Barendregt. Introduction to generalized type systems. *J. Functional Programming*, 1(2):125–154, 1991.

[BKK$^+$98] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.

[BKVV05] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. *Stratego/XT Tutorial, Examples, and Reference Manual.* Department of Information and Computing Sciences, Universiteit Utrecht, August 2005. (Draft).

[BvEG$^+$87] Hendrik Pieter Barendregt, Marko C. J. D. van Eekelen, John R. W. Glauert, Richard Kennaway, Marinus J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In *PARLE (2)*, volume 259 of *LNCS*, pages 141–158. Springer, 1987.

[CDE$^+$02] 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.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82*, pages 207–212. ACM, 1982.

[Eke96] Steven Eker. Fast matching in combinations of regular equational theories. In *WRLA '96*, volume 4 of *ENTCS*, pages 90–109, 1996.

[FH92] Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *J. TCS*, 103(2):235–271, 1992.

[FPST07] Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *PEPM '07*, pages 112–121. ACM, 2007.

[Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and validation methods*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[GWM$^+$00] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action.* Kluwer, 2000.

[HdM] Yorck Hunke and Oege de Moor. Aiding dependent type checking with rewrite rules.

[HSG98] Ravi Hosabettu, Mandayam K. Srivas, and Ganesh Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In *CAV '98*, volume 1427 of *LNCS*, pages 122–134. Springer, 1998.

[KK00] Fairouz Kamareddine and Jan Willem Klop, editors. *Special issue on Type Theory and Term Rewriting: a collection of papers*, volume 10(3). Oxford University Press, 2000.

[Klo92] Jan Willem 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.

[KMF07] George Kuan, David MacQueen, and Robert Bruce Findler. A rewriting semantics for type inference. In *ESOP '07*, volume 4421 of *LNCS*, pages 426–440. Springer, 2007.

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *TOPLAS*, 28(4):619–695, 2006.

[KSK06] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. A PVS based framework for validating compiler optimizations. In *SEFM '06*, pages 108–117. IEEE Computer Society, 2006.

[KSK07] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. Structuring optimizing transformations and proving them sound. In *COCV '06*, volume 176(3) of *ENTCS*, pages 79–95. Elsevier, 2007.

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In *POPL '07*, pages 173–184. ACM, 2007.

[LP03] Michael Y. Levin and Benjamin C. Pierce. Tinkertype: a language for playing with formal systems. *J. Functional Programing*, 13(2):295–316, 2003.

[LW91] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *POPL '91*, pages 291–302. ACM, 1991.

[Mam07] Azamatbek Mametjanov. Types and program transformations. In *OOPSLA '07 Companion*, pages 937–938. ACM, 2007.

[Mes92] José Meseguer. Conditioned rewriting logic as a unified model of concurrency. *J. TCS*, 96(1):73–155, 1992.

[MFFF04] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *RTA '04*, volume 3091 of *LNCS*, pages 301–311. Springer, 2004.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17(3):348–375, 1978.

[MR04] José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In *IJCAR '04*, volume 3097 of *LNCS*, pages 1–44. Springer, 2004.

[MR07] José Meseguer and Grigore Rosu. The rewriting logic semantics project. *J. TCS*, 373(3):213–237, 2007. Also appeared in *SOS '05*, volume 156(1) of *ENTCS*, pages 27–56, 2006.

[Plo04] Gordon 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.

[Plu98] Detlef Plump. Term graph rewriting, 1998.

[Ros06] Grigore Rosu. 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.

[SM04] Mark-Oliver Stehr and José 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.

[SW00] Christopher Strachey and Christopher P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. *Higher-Order and Symb. Computation*, 13(1/2):135–152, 2000.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[vdBHKO02] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.

[Vis03] Eelco Visser. Program Transf. with Stratego/XT: Rules, Strategies, Tools, and Systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.