Electronic Notes in Theoretical Computer Science 75 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume75.html 14 pages

Syntactic Type Soundness for HM(X)

Christian Skalka [1]

The University of Vermont

François Pottier [2]

INRIA Rocquencourt

Abstract

The HM(X) framework is a constraint-based type framework with built-in let-polymorphism. This paper establishes purely syntactic type soundness for the framework, treating an extended version of the language containing state and recursive binding. These results demonstrate that any instance of HM(X), comprising a specialized constraint system and possibly additional functional constants and their types, enjoys syntactic type soundness.

1 Introduction

This paper presents a purely syntactic type soundness result for HM(X), in the style of Wright and Felleisen [7]. A soundness result based on a denotational semantics was originally presented by Odersky *et al.* [1]. Sulzmann [5] conjectured that type safety for HM(X) can be established in a syntactic way, but did not provide a proof. Recently, Pottier [2] used HM(X) as a vehicle to illustrate a *semi-syntactic* proof technique, which does not rely on a denotational semantics, yet allows part of the proof to be carried out by induction on type derivations, suppressing the need for normalization lemmas.

None of the two existing results mentioned above is adequate in the event that a purely syntactic result is wished for. Such a wish can arise, for instance, out of the desire to build a custom type system on top of an instance of HM(X) via a *type system factory.* We call type system factory a construction which, given any type system that satisfies subject reduction (and perhaps some additional properties), produces a new type system, dedicated to a specific purpose, together with its correctness proof. For instance, type system factories which produce type-based security analyses are described in [3,4].

[1] Email: skalka@cs.uvm.edu

[2] Email: Francois.Pottier@inria.fr

$$
\begin{array}{rcll}
x, z & \in & \mathit{ID} & \textit{identifiers} \\
l & \in & \mathit{Loc} & \textit{memory locations} \\
\mathbf{c} & \in & \mathit{Const} & \textit{constants} \\
v & ::= & x \mid l \mid \mathsf{fix}\, z.\lambda x.e \mid \mathrm{ref} \mid := \mid (:= l) \mid \,!\, \mid \mathbf{c} & \textit{values} \\
e & ::= & v \mid e\, e \mid \mathsf{let}\, x = v \,\mathsf{in}\, e & \textit{expressions} \\
E & ::= & [\,] \mid E\, e \mid v\, E & \textit{evaluation contexts}
\end{array}
$$

Fig. 1. Language grammar for HM(X)

This presentation of HM(X) extends previous results by treating a version of the core language that contains state and a primitive recursive binding mechanism. The addition of state increases the expressivity of the programming language. A primitive recursive binding mechanism is a welcome convenience; previously, it was necessary to either define a fixpoint combinator, or introduce one as a constant, entailing additional proof overhead to obtain type soundness for an instance of the framework.

Our presentation of HM(X) is otherwise identical to that of [1,5]. The main difference resides in our axiomatization of the meaning of constraints, which is more direct; see section 2.2. Our proof technique is standard, following Wright and Felleisen [7]. The central results are subject reduction, progress, and type safety for the HM(X) framework, stated and proved in section 4.

2 Definitions

In this section we present the HM(X) framework, that is, the programming language and its type system.

2.1 The Language

The core language is a call-by-value functional calculus, extended with a recursive binding mechanism built into function definitions, and mechanisms for state. We postulate countably infinite sets of identifiers, locations, and constants. The language grammar is defined in figure 1. Note that, following [6], we impose a *value restriction* on let bindings, precluding unsafe interaction between imperative features and polymorphism.

The operational semantics is defined on *configurations* e/ς, where a *store* ς is a partial mapping from locations to values. We write $\varsigma[l \mapsto v]$ to denote the store which maps l to v and otherwise agrees with ς. The empty store is denoted $\varnothing$. The one-step reduction rules for HM(X) are then defined in figure 2. We write $\rightarrow^\star$ to denote the reflexive, transitive closure of $\rightarrow$. The

$$
\begin{array}{rcl l r}
(\mathsf{fix}\, z.\lambda x.e)v/\varsigma & \rightarrow & e[v/x][\mathsf{fix}\, z.\lambda x.e/z]/\varsigma & & (\beta) \\
\mathsf{let}\, x = v\, \mathsf{in}\, e/\varsigma & \rightarrow & e[v/x]/\varsigma & & (\mathit{let}) \\
\mathrm{ref}\, v/\varsigma & \rightarrow & l/\varsigma[l \mapsto v] & l \notin \mathrm{dom}(\varsigma) & (\mathit{ref}) \\
:=\, l\, v/\varsigma & \rightarrow & v/\varsigma[l \mapsto v] & l \in \mathrm{dom}(\varsigma) & (\mathit{assign}) \\
!\, l/\varsigma & \rightarrow & \varsigma(l)/\varsigma & & (\mathit{deref}) \\
\mathbf{c}\, v/\varsigma & \rightarrow & \delta(\mathbf{c}, v)/\varsigma & & (\delta) \\
E[e]/\varsigma & \rightarrow & E[e']/\varsigma' & \text{where } e/\varsigma \rightarrow e'/\varsigma' & (\mathit{context})
\end{array}
$$

Fig. 2. Operational semantics for HM(X)

interpretation of constants is given by a (possibly partial) function δ which maps a pair of a constant and a closed value to a closed value.

2.2 Constraint Systems

Any instance of the HM(X) framework is parameterized by a *constraint system.* This system must at least comprise the following language of types and constraints, where $\mathcal{V}$ is a countably infinite set of type variables:

$$
\begin{array}{ll r}
\alpha, \beta \in \mathcal{V} & & \textit{type variables} \\
\tau \ ::= \alpha \mid \tau \rightarrow \tau \mid \tau\ \mathrm{ref} \mid \ldots & & \textit{types} \\
C \ ::= \mathbf{true} \mid \tau = \tau \mid \tau \leq \tau \mid C \wedge C \mid \exists \alpha . C \mid \ldots & & \textit{constraints}
\end{array}
$$

To interpret constraints, we adopt the model-based approach described in [2], which is established via a mapping from types into a universe of partially ordered monotypes T.

Definition 2.1 [Model] Let $(T, \leq)$ be a partially ordered set, where $t \in T$ is called a *monotype.* Let $\rightarrow$ be a function from $T \times T$ into T, where $t_1 \rightarrow t_2 \leq t'_1 \rightarrow t'_2$ implies $t'_1 \leq t_1$ and $t_2 \leq t'_2$. Let ref be a function from T to T, such that t ref $\leq t'$ ref implies $t = t'$. We require $t_1\,\mathrm{ref} \leq t_2 \rightarrow t_3$ and $t_2 \rightarrow t_3 \leq t_1\,\mathrm{ref}$ to be false for any $t_1, t_2, t_3 \in T$.

Definition 2.2 [Interpretation] An *assignment* ρ is a total mapping from $\mathcal{V}$ to T. An *interpretation* of a constraint system consists of an extension of assignments to arbitrary types, and a *constraint satisfaction relation*, denoted

$\rho \vdash C$. The interpretation is *standard* iff the following conditions are satisfied:

$$\begin{array}{rcl} \rho(\tau_1 \rightarrow \tau_2) & = & \rho(\tau_1) \rightarrow \rho(\tau_2) \\ \rho(\tau \text{ ref}) & = & \rho(\tau) \text{ ref} \end{array}$$

$$\begin{array}{rcl} & \rho \vdash \mathbf{true} & \\ \rho \vdash \tau_1 = \tau_2 & \Leftrightarrow & \rho(\tau_1) = \rho(\tau_2) \\ \rho \vdash \tau_1 \leq \tau_2 & \Leftrightarrow & \rho(\tau_1) \leq \rho(\tau_2) \\ \rho \vdash C_1 \wedge C_2 & \Leftrightarrow & (\rho \vdash C_1) \wedge (\rho \vdash C_2) \\ \rho \vdash \exists \alpha . C & \Leftrightarrow & \exists t . \rho[\alpha \mapsto t] \vdash C \end{array}$$

If $\rho \vdash C$ holds, we say that ρ *satisfies* or is a *solution* of C. We write $C \Vdash C'$ iff every solution of C is also a solution of C'.

We identify constraints modulo logical equivalence, that is, we identify C and D when $C \Vdash D$ and $D \Vdash C$ hold. A variable α is deemed *free* in a constraint C iff $C \neq \exists \alpha . C$. We write $\mathrm{fv}(C)$ for the set of all variables free in C.

Our presentation differs from that of Odersky *et al.* [1] by viewing constraints as formulae interpreted in T, rather than as elements of an abstract *cylindric constraint system*. Our presentation is thus perhaps slightly less general, but more concise. Also, we abandon Odersky *et al.*'s notion of constraints in *solved form*. Instead, we identify constraints modulo logical equivalence, which means that we do not care about their syntactic representation. We believe that the representation of constraints is an important issue when designing a constraint solver, but is irrelevant when proving the type system correct.

2.3 The Type System

The HM(X) type system is defined as a system of deduction rules, given in figure 3, whose consequents are *judgements* of the form $C, \Gamma \vdash e : \sigma$ where C is a constraint, Γ is a *type environment*, and σ is a *type scheme*. These notions are introduced in the following definition:

Definition 2.3 *Type schemes* are of the form $\forall \bar{\alpha}[C].\tau$. Abusing notation, we abbreviate a scheme $\forall \varnothing[\mathbf{true}].\tau$ as τ, and abbreviate $\forall \bar{\alpha}[\mathbf{true}].\tau$ as $\forall \bar{\alpha}.\tau$. We identify type schemes modulo α-equivalence. *Type environments* Γ are sequences of bindings of the form $x : \sigma$ and $l : \tau$.

A type scheme σ is *consistent* with respect to a constraint C if C guarantees that σ has at least one instance. This notion, defined below, appears as a technical side-condition in rule Var. This extra side-condition is our only

$$\textsc{Var}\quad \frac{\Gamma(x) = \sigma \qquad C \Vdash \sigma}{C, \Gamma \vdash x : \sigma} \qquad\qquad \textsc{Loc}\quad \frac{\Gamma(l) = \tau}{C, \Gamma \vdash l : \tau \text{ ref}} \qquad\qquad \textsc{Const}\quad C, \Gamma \vdash \mathbf{c} : \Delta(\mathbf{c})$$

$$\textsc{Abs}\quad \frac{C, (\Gamma; x : \tau; z : \tau \to \tau') \vdash e : \tau'}{C, \Gamma \vdash \mathsf{fix}\, z.\lambda x.e : \tau \to \tau'} \qquad\qquad \textsc{App}\quad \frac{C, \Gamma \vdash e_1 : \tau_2 \to \tau \qquad C, \Gamma \vdash e_2 : \tau_2}{C, \Gamma \vdash e_1\, e_2 : \tau}$$

$$\textsc{Ref}\quad C, \Gamma \vdash \text{ref} : \forall \alpha.\alpha \to \alpha \text{ ref} \qquad\qquad \textsc{Assign}\quad C, \Gamma \vdash := : \forall \alpha.\alpha \text{ ref} \to \alpha \to \alpha$$

$$\textsc{Deref}\quad C, \Gamma \vdash\, ! : \forall \alpha.\alpha \text{ ref} \to \alpha \qquad\qquad \textsc{Let}\quad \frac{C, \Gamma \vdash v : \sigma \qquad C, (\Gamma; x : \sigma) \vdash e : \tau}{C, \Gamma \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e : \tau}$$

$$\textsc{Sub}\quad \frac{C, \Gamma \vdash e : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash e : \tau'} \qquad\qquad \forall\ \textsc{Intro}\quad \frac{C \wedge D, \Gamma \vdash v : \tau \qquad \bar{\alpha} \cap \text{fv}(C, \Gamma) = \varnothing}{C \wedge \exists \bar{\alpha}.D, \Gamma \vdash v : \forall \bar{\alpha}[D].\tau}$$

$$\forall\ \textsc{Elim}\quad \frac{C, \Gamma \vdash v : \forall \bar{\alpha}[D].\tau \qquad C \Vdash [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash v : [\bar{\tau}/\bar{\alpha}]\tau}$$

Fig. 3. The system HM(X)

deviation from the rules given in [1,5]. Its effect is to allow some theorems to be stated without a "consistency" requirement on Γ.

Definition 2.4 We say that a type scheme $\sigma = \forall \bar{\alpha}[D].\tau$ is consistent with respect to a constraint C, and we write $C \Vdash \sigma$, iff $C \Vdash \exists \bar{\alpha}.D$. We say that σ is consistent iff $\mathbf{true} \Vdash \sigma$.

Let Δ be a fixed total mapping from the constants to closed, consistent type schemes. Δ is looked up in rule Const to associate a type scheme with a constant.

Definition 2.5 A judgement $C, \Gamma \vdash e : \sigma$ is *valid* (or *holds*) iff it is derivable according to the rules of figure 3 and C is satisfiable. Then, e is *well-typed.*

It is straightforward to check that, if $C, \Gamma \vdash e : \sigma$ is derivable, then $C \Vdash \sigma$ holds. This explains why the well-typedness of e can be determined by checking whether C alone is satisfiable; there is no need to inspect σ in addition.

For the type system to be safe, the semantics of constants, given by δ, must be correctly approximated by their types, given by Δ.

Definition 2.6 [δ-Typability] Let C be satisfiable. We require that, for every constant $\mathbf{c}$ and closed value v, if $C, \Gamma \vdash \mathbf{c} : \tau_1 \rightarrow \tau_2$ and $C, \Gamma \vdash v : \tau_1$ hold, then $\delta(\mathbf{c}, v)$ is defined and $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ holds. We also require $C, \Gamma \vdash \mathbf{c} : \tau \operatorname{ref}$ to *not* hold.

The following definition sums up the requirements that bear on every instance of the parameterized type system HM(X).

Definition 2.7 An instance of HM(X) is defined by

- an extension of the type and constraint language, together with a standard interpretation, as specified in definitions 2.1 and 2.2;
- a particular choice of the set of constants *Const*, together with functions δ and Δ, meeting the δ-typability requirement of definition 2.6.

As will be proven in section 4, any such instance of HM(X) enjoys syntactic type safety.

3 Preliminary results

3.1 Type substitutions

Sulzmann [5] gives two equivalent versions of the HM(X) type rules. In the one shown here, rule ∀ Elim allows the universally quantified type variables to be instantiated using an arbitrary substitution. In the other version, not shown in this paper, rule ∀ Elim requires these variables to be instantiated with the identity substitution, but a new rule appears (∃-Intro) which allows arbitrary substitutions to be encoded within a constraint. The two presentations are equivalent, that is, they give rise to the same valid judgements. As a result, it is enough to prove one of them correct.

Here, we adopt the substitution-based version. Accordingly, we must now demonstrate a series of results related to substitutions.

Definition 3.1 A *substitution* φ is a finite mapping from type variables to types. A *renaming* ϱ is a bijective mapping from a finite set of type variables to itself. Substitutions and renamings are extended to total mappings from types to types, from constraints to constraints, and from type schemes to type schemes, in the natural, capture-avoiding manner.

Lemma 3.2 *If $C \Vdash D$ then $\varphi(C) \Vdash \varphi(D)$. If $C \Vdash \sigma$, then $\varphi(C) \Vdash \varphi(\sigma)$.*

Lemma 3.3 *If φ_1 is idempotent and* $\operatorname{dom}(\varphi_2)$ *and* $\operatorname{fv}(\operatorname{rng}(\varphi_1)) \cup \operatorname{dom}(\varphi_1)$ *are disjoint then $\varphi_1 \circ \varphi_2 \circ \varphi_1 = \varphi_1 \circ \varphi_2$.*

Lemma 3.4 (Type Instantiation) *If there exists a derivation of $C, \Gamma \vdash e : \sigma$, then there exists a derivation of $\varphi(C), \varphi(\Gamma) \vdash e : \varphi(\sigma)$ with the same structure.*

Proof. By induction on the input derivation. We give only the key cases and follow the notations of figure 3. Note that the structure of the derivation is preserved by construction in the proof.

Cases Var, Sub. By induction hypothesis and by lemma 3.2.

Case $\forall$ Intro. Without loss of generality, we may require $\bar{\alpha} \cap \mathrm{fv}(\mathrm{rng}(\varphi)) = \bar{\alpha} \cap \mathrm{dom}(\varphi) = \varnothing$. Indeed, if such were not the case, one could apply the induction hypothesis to the premise and to a renaming which maps $\bar{\alpha}$ to fresh variables and does not affect any other variable free in the premise. Because the variables $\bar{\alpha}$ do not appear free in the conclusion, the latter would remain unchanged.

Now, let us apply the induction hypothesis to the premise and φ. This yields $\varphi(C) \wedge \varphi(D), \varphi(\Gamma) \vdash e : \varphi(\tau)$. From $\bar{\alpha} \cap \mathrm{fv}(C, \Gamma) = \varnothing$ and the above requirement, we deduce $\bar{\alpha} \cap \mathrm{fv}(\varphi(C), \varphi(\Gamma)) = \varnothing$. Thus, we may apply $\forall$ Intro, which yields $\varphi(C) \wedge \exists\bar{\alpha}.\varphi(D), \varphi(\Gamma) \vdash e : \forall\bar{\alpha}[\varphi(D)].\varphi(\tau)$. Again, thanks to the above requirement, this is $\varphi(C \wedge \exists\bar{\alpha}.D), \varphi(\Gamma) \vdash e : \varphi(\forall\bar{\alpha}[D].\tau)$.

Case $\forall$ Elim. Every substitution is the composition of an idempotent substitution and a renaming. Thus, we consider two sub-cases.

First, let us assume that φ is idempotent. By the induction hypothesis, we have $\varphi(C), \varphi(\Gamma) \vdash e : \varphi(\forall\bar{\alpha}[D].\tau)$. Without loss of generality, we may assume that $\bar{\alpha} \cap \mathrm{fv}(\mathrm{rng}(\varphi)) = \varnothing$ and $\bar{\alpha} \cap \mathrm{dom}(\varphi) = \varnothing$. (This follows from the fact that we identify type schemes modulo α-equivalence.) This yields $\varphi(C), \varphi(\Gamma) \vdash e : \forall\bar{\alpha}[\varphi(D)].\varphi(\tau)$ and (by lemma 3.3) $\varphi \circ [\bar{\tau}/\bar{\alpha}] \circ \varphi = \varphi \circ [\bar{\tau}/\bar{\alpha}]$. Now, lemma 3.2 yields $\varphi(C) \Vdash \varphi([\bar{\tau}/\bar{\alpha}]D)$, that is, $\varphi(C) \Vdash \varphi \circ [\bar{\tau}/\bar{\alpha}](\varphi(D))$. Therefore, by $\forall$ Elim, we obtain $\varphi(C), \varphi(\Gamma) \vdash e : \varphi \circ [\bar{\tau}/\bar{\alpha}](\varphi(\tau))$, that is, $\varphi(C), \varphi(\Gamma) \vdash e : \varphi([\bar{\tau}/\bar{\alpha}]\tau)$.

Second, let us assume that φ is a renaming ϱ. By applying the induction hypothesis to the premise, we obtain $\varrho C, \varrho\Gamma \vdash e : \varrho(\forall\bar{\alpha}[D].\tau)$, which can be written $\varrho C, \varrho\Gamma \vdash e : \forall(\varrho\bar{\alpha})[\varrho D].\varrho\tau$. Furthermore, lemma 3.2 yields $\varrho C \Vdash \varrho[\bar{\tau}/\bar{\alpha}]D$, that is, $\varrho C \Vdash [\varrho\bar{\tau}/\varrho\bar{\alpha}]\varrho D$. Then, $\forall$ Elim, applied to the substitution $[\varrho\bar{\tau}/\varrho\bar{\alpha}]$, yields $\varrho C, \varrho\Gamma \vdash e : [\varrho\bar{\tau}/\varrho\bar{\alpha}]\varrho\tau$, that is, $\varrho C, \varrho\Gamma \vdash e : \varrho[\bar{\tau}/\bar{\alpha}]\tau$. □

3.2 Normalization

In this section we define a normalized form for HM(X) type derivations. This normalization provides for a much easier analysis of type derivations in the subject reduction proof.

Lemma 3.5 *If* $\mathrm{dom}(\varphi) \subseteq \bar{\alpha}$ *then* $\varphi(C) \Vdash \exists\bar{\alpha}.C$.

Lemma 3.6 *Any two consecutive instances of* $\forall$ Intro *and* $\forall$ Elim *may be suppressed.*

Proof. Suppose the following sequence appears in a derivation:

$$\dfrac{\dfrac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C, \Gamma) = \varnothing}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau}\ (\forall\ \textsc{Intro}) \qquad C \wedge \exists\bar{\alpha}.D \Vdash [\bar{\tau}/\bar{\alpha}]D}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau}\ (\forall\ \textsc{Elim})$$

From $C \wedge \exists\bar{\alpha}.D \Vdash [\bar{\tau}/\bar{\alpha}]D$, we may deduce $C \wedge \exists\bar{\alpha}.D \Vdash C \wedge [\bar{\tau}/\bar{\alpha}]D$. However, by lemma 3.5, we have $[\bar{\tau}/\bar{\alpha}]D \Vdash \exists\bar{\alpha}.D$, so $C \wedge \exists\bar{\alpha}.D$ and $C \wedge [\bar{\tau}/\bar{\alpha}]D$ are equivalent. Furthermore, considering $\bar{\alpha} \cap \mathrm{fv}(C) = \varnothing$, we have $C \wedge [\bar{\tau}/\bar{\alpha}]D = [\bar{\tau}/\bar{\alpha}](C \wedge D)$. Similarly, $\bar{\alpha} \cap \mathrm{fv}(\Gamma) = \varnothing$ implies $[\bar{\tau}/\bar{\alpha}]\Gamma = \Gamma$. Now, lemma 3.4, applied to the upper left judgement, yields $[\bar{\tau}/\bar{\alpha}](C \wedge D), [\bar{\tau}/\bar{\alpha}]\Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau$, which, according to the above arguments, is $C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau$. The derivation of this judgement has the same structure as that of the upper left judgement, so these instances of ∀ Intro and ∀ Elim have effectively been suppressed. □

Lemma 3.7 (Normalization) *If $C, \Gamma \vdash e : \tau$ holds, then it must follow by* Sub *from a judgement $\mathcal{J}$ such that*

(i) *if e is $\mathsf{let}\, x = v \,\mathsf{in}\, e'$ then $\mathcal{J}$ follows by* Let*;*

(ii) *if e is $\mathsf{fix}\, z.\lambda x.e'$ then $\mathcal{J}$ follows by* Abs*;*

(iii) *if e is $e_1\, e_2$ then $\mathcal{J}$ follows by* App*;*

(iv) *if e is l then $\mathcal{J}$ follows by* Loc*;*

(v) *if e is x then $\mathcal{J}$ follows by* Var *and* ∀ Elim*;*

(vi) *if e is* **c** *then $\mathcal{J}$ follows by* Const *and* ∀ Elim*;*

(vii) *if e is* ref *then $\mathcal{J}$ follows by* Ref *and* ∀ Elim*;*

(viii) *if e is* ! *then $\mathcal{J}$ follows by* Deref *and* ∀ Elim*;*

(ix) *if e is* := *then $\mathcal{J}$ follows by* Assign *and* ∀ Elim*.*

Proof. The judgement $C, \Gamma \vdash e : \tau$ must be the consequence of a syntax-directed rule, possibly followed by a sequence of instances of Sub, ∀ Elim and ∀ Intro.

By construction, ∀ Intro cannot be followed by itself or by Sub. Lemma 3.6 shows that ∀ Intro need never be followed by ∀ Elim. Lastly, given the form of the judgement at hand, ∀ Intro cannot be the last rule in the derivation. It follows that ∀ Intro need not appear at all in the sequence.

By construction, ∀ Elim cannot follow itself or Sub, so the sequence must consist of at most one instance of ∀ Elim, followed by a number of instances of Sub. By reflexivity and transitivity of entailment, the latter may be expanded or reduced to a single instance of Sub.

To conclude, notice that ∀ Elim cannot follow Loc, Let, Abs or App. □

3.3 Value Substitution

In this section, we establish a classic *substitution* lemma, which will be at the heart of the β- and let-reduction cases in the subject reduction proof. We begin with a *weakening* lemma, which shows that a valid judgement remains valid under a stronger constraint.

Lemma 3.8 (Weakening) *$C, \Gamma \vdash e : \sigma$ and $C' \Vdash C$ imply $C', \Gamma \vdash e : \sigma$.*

Proof. By induction on the input derivation. We give only the key cases and follow the notations of figure 3.

Cases Var, Sub and ∀ Elim follow by transitivity of entailment.

Case ∀ Intro. We have a deduction of the form

$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C, \Gamma) = \varnothing}{C \wedge \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau}$$

Without loss of generality, we may assume $\bar{\alpha} \cap \mathrm{fv}(C') = \varnothing$; if this were not the case, we could apply lemma 3.4 to the first premise to make it so. Now, clearly $C' \wedge C \wedge D \Vdash C \wedge D$, so the induction hypothesis yields $C' \wedge C \wedge D, \Gamma \vdash e : \tau$. Furthermore, we have $\bar{\alpha} \cap \mathrm{fv}(C' \wedge C, \Gamma) = \varnothing$, therefore ∀ Intro yields $C' \wedge C \wedge \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau$. Lastly, by assumption, we have $C' \Vdash C \wedge \exists \bar{\alpha}.D$, so $C' = C' \wedge C \wedge \exists \bar{\alpha}.D$, therefore $C', \Gamma \vdash e : \forall \bar{\alpha}[D].\tau$ holds. □

Lemma 3.9 (Substitution) *If $C, \Gamma; x : \sigma' \vdash e : \sigma$ and $C, \Gamma \vdash v : \sigma'$ then $C, \Gamma \vdash e[v/x] : \sigma$.*

Proof. By induction on the derivation of $C, \Gamma; x : \sigma' \vdash e : \sigma$. We give only the key cases.

Case ∀ Intro. In this case $\sigma = \forall \bar{\alpha}[D].\tau$, $C = C' \wedge \exists \bar{\alpha}.D$ and we have a deduction of the form:

$$\frac{C' \wedge D, \Gamma; x : \sigma' \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C', \Gamma; x : \sigma') = \varnothing}{C' \wedge \exists \bar{\alpha}.D, \Gamma; x : \sigma' \vdash e : \forall \bar{\alpha}[D].\tau}$$

By assumption we have that $C' \wedge \exists \bar{\alpha}.D, \Gamma \vdash v : \sigma'$ holds, and clearly $C' \wedge D \Vdash C' \wedge \exists \bar{\alpha}.D$, therefore by lemma 3.8 we have $C' \wedge D, \Gamma \vdash v : \sigma'$. Then, by the induction hypothesis, $C' \wedge D, \Gamma \vdash e[v/x] : \tau$ holds. The result follows by ∀ Intro.

Case Var. Suppose that $e = x' \neq x$. Then $e[v/x] = e$ and $\Gamma(x') = (\Gamma; x : \sigma')(x')$, so the lemma holds by Var. Suppose on the other hand that $e = x$; then $e[v/x] = v$, so the lemma holds by assumption.

Case Let. In this case $e = \mathsf{let}\, x' = v' \,\mathsf{in}\, e'$, $\sigma = \tau$ and we have a deduction of the following form:

$$\frac{C, \Gamma; x : \sigma' \vdash v' : \sigma'' \qquad C, \Gamma; x : \sigma'; x' : \sigma'' \vdash e' : \tau}{C, \Gamma; x : \sigma' \vdash \mathsf{let}\, x' = v' \,\mathsf{in}\, e' : \tau}$$

By the induction hypothesis we have $C, \Gamma \vdash v'[v/x] : \sigma''$; and supposing that $x \neq x'$ it is the case that $\Gamma; x : \sigma'; x' : \sigma'' = \Gamma; x' : \sigma''; x : \sigma'$, hence we have also $C, \Gamma; x' : \sigma'' \vdash e'[v/x] : \tau$ by the induction hypothesis, so that $C, \Gamma \vdash \mathsf{let}\, x' = v'[v/x] \,\mathsf{in}\, e'[v/x] : \tau$ by Let, hence $C, \Gamma \vdash (\mathsf{let}\, x' = v' \,\mathsf{in}\, e')[v/x] : \tau$ by definition. On the other hand, if $x = x'$ then $\Gamma; x : \sigma'; x' : \sigma'' = \Gamma; x' : \sigma''$, so that $C, \Gamma; x' : \sigma'' \vdash e' : \tau$ by assumption, and since $C, \Gamma \vdash v'[v/x] : \sigma''$ by the preceding, the judgement $C, \Gamma \vdash \mathsf{let}\, x' = v'[v/x] \,\mathsf{in}\, e' : \tau$ holds by Let, therefore $C, \Gamma \vdash (\mathsf{let}\, x' = v' \,\mathsf{in}\, e')[v/x] : \tau$ by definition.

Case Abs. In this case $e = \mathsf{fix}\, z.\lambda x'.e'$, $\sigma = \tau_1 \to \tau_2$ and we have a deduction of the following form:

$$\frac{C, \Gamma; x : \sigma'; x' : \tau_1; z : \tau_1 \to \tau_2 \vdash e' : \tau_2}{C, \Gamma; x : \sigma' \vdash \mathsf{fix}\, z.\lambda x'.e' : \tau_1 \to \tau_2}$$

Supposing that $x \neq x'$ and $x \neq z$ it is the case that

$$\Gamma; x : \sigma'; x' : \tau_1; z : \tau_1 \to \tau_2 = \Gamma; x' : \tau_1; z : \tau_1 \to \tau_2; x : \sigma'$$

hence we have $C, \Gamma; x' : \tau_1; z : \tau_1 \to \tau_2 \vdash e'[v/x] : \tau_2$ by the induction hypothesis, so $C, \Gamma \vdash \mathsf{fix}\, z.\lambda x'.(e'[v/x]) : \tau_1 \to \tau_2$ by Abs, therefore $C, \Gamma \vdash (\mathsf{fix}\, z.\lambda x'.e')[v/x] : \tau_1 \to \tau_2$ by definition. On the other hand, supposing that $x = x'$ it is the case that

$$\Gamma; x : \sigma'; x' : \tau_1; z : \tau_1 \to \tau_2 = \Gamma; x' : \tau_1; z : \tau_1 \to \tau_2$$

and since $C, (\Gamma; x : \sigma'; x' : \tau_1; z : \tau_1 \to \tau_2) \vdash e' : \tau_2$ by assumption therefore $C, (\Gamma; x' : \tau_1; z : \tau_1 \to \tau_2) \vdash e' : \tau_2$, so $C, \Gamma \vdash \mathsf{fix}\, z.\lambda x'.e' : \tau_1 \to \tau_2$ by Abs, thus $C, \Gamma \vdash (\mathsf{fix}\, z.\lambda x'.e')[v/x] : \tau_1 \to \tau_2$ by definition. The case in which $x = z$ follows similarly. □

Lemma 3.10 (Substitution for functions) *Let $\Gamma' = (\Gamma; x : \tau'; z : \tau' \to \tau)$. If $C, \Gamma' \vdash e : \tau$ and $C, \Gamma \vdash v : \tau'$, then $C, \Gamma \vdash e[v/x][\mathsf{fix}\, z.\lambda x.e/z] : \tau$.*

Proof. By Abs and two consecutive applications of lemma 3.9. □

4 Central Results

In this section we demonstrate the type soundness results for HM(X), specifically subject reduction, progress and type safety.

Definition 4.1 In order to properly state subject reduction, type judgements are extended to configurations:

Config

$$\frac{\begin{array}{c} C, \Gamma \vdash e : \tau \\ \forall l \in \mathrm{dom}(\Gamma) \quad C, \Gamma \vdash \varsigma(l) : \Gamma(l) \end{array}}{C, \Gamma \vdash e/\varsigma : \tau}$$

A configuration e/ς is *well-typed* if there exists a judgement $C, \Gamma \vdash e/\varsigma : \tau$ deducible by Config, with C satisfiable; such a judgement is *valid.*

Theorem 4.2 (Subject Reduction) *Let C be satisfiable. If $C, \Gamma \vdash e_1/\varsigma_1 : \tau$ is derivable and $e_1/\varsigma_1 \to e_2/\varsigma_2$, then, for some Γ' which extends Γ with bindings for new memory locations, $C, \Gamma' \vdash e_2/\varsigma_2 : \tau$ is derivable.*

Proof. By induction on the definition of the reduction relation (see figure 2).

According to lemma 3.7, the derivation of $C, \Gamma \vdash e_1 : \tau$ ends with an instance of Sub, which we will disregard, without loss of generality. (Indeed, we then have $C, \Gamma \vdash e_1 : \tau'$ and $C \Vdash \tau' \leq \tau$; once we have proven $C, \Gamma \vdash e_2 : \tau'$, applying Sub again shall yield $C, \Gamma \vdash e_2 : \tau$, as desired.)

For reduction cases which do not affect the store, it is sufficient to prove that $C, \Gamma \vdash e_2 : \tau$ is derivable to demonstrate the result.

Case (δ). Then, e_1 is $\mathbf{c}\, v$ and e_2 is $\delta(\mathbf{c}, v)$. By lemma 3.7 we have a sub-derivation of the following form:

$$\frac{C, \Gamma \vdash \mathbf{c} : \tau_1 \to \tau \qquad C, \Gamma \vdash v : \tau_1}{C, \Gamma \vdash \mathbf{c}\, v : \tau}$$

Then, according to definition 2.6, $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau$ holds.

Case (β). Then, e_1 is $(\mathsf{fix}\, z.\lambda x.e)\, v$ and e_2 is $e[v/x][\mathsf{fix}\, z.\lambda x.e/z]$. By lemma 3.7 we have a sub-derivation of the following form:

$$\dfrac{\dfrac{\dfrac{C, \Gamma; x : \tau_1'; z : \tau_1' \to \tau' \vdash e : \tau'}{C, \Gamma \vdash \mathsf{fix}\, z.\lambda x.e : \tau_1' \to \tau'} \qquad C \Vdash \tau_1' \to \tau' \leq \tau_1 \to \tau}{C, \Gamma \vdash \mathsf{fix}\, z.\lambda x.e : \tau_1 \to \tau} \qquad C, \Gamma \vdash v : \tau_1}{C, \Gamma \vdash (\mathsf{fix}\, z.\lambda x.e)\, v : \tau}$$

Now, $C \Vdash \tau_1' \to \tau' \leq \tau_1 \to \tau$ implies $C \Vdash \tau_1 \leq \tau_1'$ and $C \Vdash \tau' \leq \tau$. Therefore $C, \Gamma \vdash v : \tau_1'$ by assumption and Sub; and since $C, (\Gamma; x : \tau_1'; z : \tau_1' \to \tau') \vdash e : \tau'$ by assumption, therefore $C, \Gamma \vdash e[v/x][\mathsf{fix}\, z.\lambda x.e/z] : \tau'$ by lemma 3.10. By Sub, $C, \Gamma \vdash e[v/x][\mathsf{fix}\, z.\lambda x.e/z] : \tau$ follows.

Case (*let*). Then, e_1 is $\mathsf{let}\, x = v\, \mathsf{in}\, e$ and e_2 is $e[v/x]$. By lemma 3.7 we have a sub-derivation of the following form:

$$\frac{C, \Gamma; x : \sigma \vdash e : \tau \qquad C, \Gamma \vdash v : \sigma}{C, \Gamma \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau}$$

By lemma 3.9, we obtain $C, \Gamma \vdash e[v/x] : \tau$.

Case (*deref*). Then, e_1 is $!\, l$ and e_2 is $\varsigma_1(l)$. By lemma 3.7, we have a sub-derivation of the following form:

$$\dfrac{\dfrac{\begin{array}{c} C, \Gamma \vdash \,! : \tau'\, \mathrm{ref} \to \tau' \\ C \Vdash \tau'\, \mathrm{ref} \to \tau' \leq \tau_1\, \mathrm{ref} \to \tau \end{array}}{C, \Gamma \vdash \,! : \tau_1\, \mathrm{ref} \to \tau} \qquad \dfrac{\dfrac{\Gamma(l) = \tau''}{C, \Gamma \vdash l : \tau''\, \mathrm{ref}} \qquad C \Vdash \tau''\, \mathrm{ref} \leq \tau_1\, \mathrm{ref}}{C, \Gamma \vdash l : \tau_1\, \mathrm{ref}}}{C, \Gamma \vdash \,!\, l : \tau}$$

By Config, $C, \Gamma \vdash \varsigma(l) : \tau''$ is derivable. and by properties of $\leq$ we have $C \Vdash \tau_1 \leq \tau$ and $C \Vdash \tau'' \leq \tau_1$. Thus, by transitivity of $\leq$ we have $C \Vdash \tau'' \leq \tau$, so $C, \Gamma \vdash \varsigma(l) : \tau$ can be derived by Sub.

Case (*ref*). The reduction is $\mathrm{ref}\, v/\varsigma_1 \longrightarrow l/\varsigma_1[l \mapsto v]$, where $l \notin \mathrm{dom}(\varsigma_1)$. By lemma 3.7 we have a sub-derivation of the following form:

$$\dfrac{\dfrac{C,\Gamma \vdash \mathrm{ref} : \tau' \to \tau'\,\mathrm{ref} \qquad C \Vdash \tau' \to \tau'\,\mathrm{ref} \leq \tau_2 \to \tau}{C,\Gamma \vdash \mathrm{ref} : \tau_2 \to \tau} \qquad C,\Gamma \vdash v : \tau_2}{C,\Gamma \vdash \mathrm{ref}\, v : \tau}$$

These imply $C \Vdash \tau_2 \leq \tau'$ and $C \Vdash \tau'\,\mathrm{ref} \leq \tau$. Define Γ' as $(\Gamma; l : \tau')$. By Loc and Sub, $C,\Gamma' \vdash l : \tau$ holds. Furthermore, since $C,\Gamma \vdash v : \tau_2$ holds and since v is $\varsigma_2(l)$, Sub yields $C,\Gamma \vdash \varsigma_2(l) : \tau'$. Because l is fresh, this implies $C,\Gamma' \vdash \varsigma_2(l) : \tau'$. Lastly, l's freshness and Config yield $C,\Gamma' \vdash l/\varsigma_2 : \tau$.

Case (*assign*). The reduction is $:= l\, v/\varsigma_1 \longrightarrow v/\varsigma_1[l \mapsto v]$, where $l \in \mathrm{dom}(\varsigma_1)$. By lemma 3.7, we have a sub-derivation of the following form:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{\begin{array}{c} C,\Gamma \vdash \,:=\, : \tau'\,\mathrm{ref} \to \tau' \to \tau' \\ C \Vdash \tau'\,\mathrm{ref} \to \tau' \to \tau' \leq \tau_1 \to \tau_2 \to \tau_3 \end{array}}{C,\Gamma \vdash \,:=\, : \tau_1 \to \tau_2 \to \tau_3} \qquad \dfrac{\begin{array}{c} \dfrac{\Gamma(l) = \tau''}{C,\Gamma \vdash l : \tau''\,\mathrm{ref}} \\ C \Vdash \tau''\,\mathrm{ref} \leq \tau_1 \end{array}}{C,\Gamma \vdash l : \tau_1}}{C,\Gamma \vdash \,:= l : \tau_2 \to \tau_3} \\ C \Vdash \tau_2 \to \tau_3 \leq \tau_2' \to \tau \end{array}}{C,\Gamma \vdash \,:= l : \tau_2' \to \tau} \qquad C,\Gamma \vdash v : \tau_2'}{C,\Gamma \vdash \,:= l\, v : \tau}$$

From these, we deduce $C \Vdash \tau_2' \leq \tau_2$ and $C \Vdash \tau_2 \leq \tau'$. Furthermore, we find $C \Vdash \tau''\,\mathrm{ref} \leq \tau_1 \leq \tau'\,\mathrm{ref}$, which implies $C \Vdash \tau' \leq \tau''$. As a result, by Sub, $C,\Gamma \vdash v : \tau''$ holds, i.e. $C,\Gamma \vdash \varsigma_2(l) : \tau''$ in this case is derivable. Furthermore, we find $C \Vdash \tau' \leq \tau_3$ and $C \Vdash \tau_3 \leq \tau$, hence $C,\Gamma \vdash v : \tau$ is derivable by Sub. The result follows by Config.

Case $E[e_1]/\varsigma_1 \longrightarrow E[e_2]/\varsigma_2$, where $e_1/\varsigma_1 \longrightarrow e_2/\varsigma_2$. This case follows by the induction hypothesis and a simple "replacement" lemma, analogous to that found in [7], except newly created memory locations must be taken into account. □

To demonstrate progress, rather than defining a class of *faulty* expressions that approximates the class of stuck expressions, and proving a *uniform evaluation* result as in e.g. [7], we adopt the more direct method of [2] and demonstrate the following:

Lemma 4.3 (Progress) *If a closed configuration e/ς is well-typed and irreducible, then e is a value.*

Proof. Suppose on the contrary that e/ς is well-typed and irreducible, but e is not a value. Then e is of the form $E[f]$, with f also well-typed as a precedent of a valid instance of Config, where one of the following cases holds:

(i) f is of the form $\mathbf{c}\,v$ and $\delta(\mathbf{c}, v)$ is undefined. Now, if $\mathbf{c}\,v$ is well-typed, then by lemma 3.7 there exists a judgement that follows by App with valid precedents $C, \Gamma \vdash \mathbf{c} : \tau_1 \to \tau_2$ and $C, \Gamma \vdash v : \tau_1$. But then by definition 2.6 it must be the case that $\delta(\mathbf{c}, v)$ is defined, which is a contradiction.

(ii) f is of the form $l\,v$. By lemma 3.7 there exists a judgement that follows by App with valid precedent $C, \Gamma \vdash l : \tau_1 \to \tau_2$. By lemma 3.7, this judgement must follow from Loc and Sub, so we have $C \Vdash \tau' \operatorname{ref} \leq \tau_1 \to \tau_2$, which is a contradiction.

(iii) f is of the form $:= v$ or $!\,v$ where v is not a memory location. In either case, by applications of lemma 3.7, we have $C, \Gamma \vdash v : \tau \operatorname{ref}$. According to definition 2.6, v cannot be a constant. One checks that all other value forms must have functional type, that is, we must have $C \Vdash \tau_1 \to \tau_2 \leq \tau \operatorname{ref}$, again a contradiction.

(iv) f is of the form $:= l\,v$ or $:= l$ and $l \notin \operatorname{dom}(\varsigma)$. f is well-typed, so $l \in \operatorname{dom}(\Gamma)$; then, Config requires $\varsigma(l)$ to be defined, a contradiction.

□

We may now state and prove progress and type safety. In order to do so, we make the usual definitions:

Definition 4.4 If $e/\varnothing \to^\star e'/\varsigma'$, where e'/ς' is irreducible but e' is not a value, then e is said to *go wrong.*

Theorem 4.5 (Type Safety) *If e is closed and well-typed, then e does not go wrong.*

Proof. Suppose that $e/\varnothing$ reduces to e'/ς' and the latter is irreducible. Since e is well-typed, there exists a derivable judgement $C, \Gamma \vdash e/\varnothing : \tau$ with C satisfiable. Then, by repeated application of theorem 4.2, we have $C, \Gamma' \vdash e'/\varsigma' : \tau$, for some Γ'. Then, by lemma 4.3, e' is a value. □

5 Conclusion

In this paper we have provided syntactic type soundness results for HM(X), including subject reduction, progress, and type safety. We have treated a version of the core language that contains features for manipulation of state, and a recursive binding mechanism. We have clearly specified the requirements that an instance of HM(X) must meet.

References

[1] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. URL: `http://www.cs.mu.oz.au/~sulzmann/publications/tapos.ps`.

[2] François Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, March 2001. URL: `ftp://ftp.inria.fr/INRIA/publication/RR/RR-4150.ps.gz`.

[3] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, Montréal, Canada, September 2000. ACM Press. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz`.

[4] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, April 2001. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-skalka-smith-esop01.ps.gz`.

[5] Martin Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000. URL: `http://www.cs.mu.oz.au/~sulzmann/publications/diss.ps.gz`.

[6] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995. URL: `http://www.cs.rice.edu/CS/PLT/Publications/lasc95-w.ps.gz`.

[7] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994. URL: `http://www.cs.rice.edu/CS/PLT/Publications/ic94-wf.ps.gz`.