A Mechanized Proof of Type Safety for the Polymorphic λ-Calculus with References*

Michalis A. Papakyriakou Prodromos E. Gerakios Nikolaos S. Papaspyrou

National Technical University of Athens
School of Electrical and Computer Engineering, Division of Computer Science,
Software Engineering Laboratory, Polytechnioupoli, 15780 Zografou, Athens, Greece
{mpapakyr, pgerakios, nickie}@softlab.ntua.gr

Abstract

In this paper we study $\lambda^{\forall,\,\mathrm{ref}}$, a Church-style typed lambda calculus with impredicative polymorphism and mutable references. We formalize the syntax, type system and call-by-value operational semantics for $\lambda^{\forall,\,\mathrm{ref}}$ in the Isabelle/HOL theorem prover and prove the type safety of the language.

1 Introduction

Type systems typically guarantee a number of interrelated safety properties, for instance, memory safety (programs can only access appropriate memory locations) and control safety (programs can only transfer control to appropriate program points). Given a formal definition of a programming language, researchers are typically interested in proving *type safety*, in the sense that the static type system precludes classes of dynamic errors when executing programs.

The polymorphic λ-calculus (usually referred to as F_2) is due to Girard and Reynolds, who independently discovered polymorphic types [1, 2]. First-class polymorphism in modern programming languages is a very useful characteristic, as it allows code re-use and modular type-checking. The language that we study in this paper is $\lambda^{\forall,\,\mathrm{ref}}$, an extension of F_2 with ML-style mutable references supporting reference allocation, dereferencing and type-preserving (weak) assignment. Type polymorphism and mutable references often interact in subtle ways leading to unsoundness. A well-known example is the problem of polymorphic references in ML [3].

Pencil-and-paper type safety proofs for programming languages of the complexity of $\lambda^{\forall,\,\mathrm{ref}}$ are very often error prone. With the aid of *proof assistants*, tools that facilitate mechanized theorem proving, researchers can be guided in such proofs and draw confidence that their proofs are correct. In this paper, we outline a mechanized type safety proof for $\lambda^{\forall,\,\mathrm{ref}}$, carried out in the Isabelle/HOL [4] proof assistant. For the representation of bound variables, we use a technique called "locally nameless" [5]. To the best of our knowledge, this is the first mechanized type safety proof for a language with polymorphic references in Isabelle/HOL and the first mechanized type safety proof, irrespective of proof assistant, for a language with references and first-class (impredicative) polymorphism.

The study of languages with polymorphic references and their metatheory is not new. Tofte proved type soundness for polymorphic references using co-induction [3]. Harper showed how a type-safety proof can be arranged so that there is no need for co-induction [6, 7]. Recent research has focused on how to best mechanize the metatheory of programming languages [8, 9] and especially to the study of fully-fledged languages, such as ML [10, 11] and Java [12].

*This paper is based on work within the research project "Theory of Algorithms and Logic: Applications in Computer Science", partially funded by the European Social Fund (75%) and the Greek Ministry of Education (25%). ΕΠΕΑΕΚ ΙΙ: Πυθαγόρας.

Syntax

$$\begin{array}{l}\tau ::= \mathsf{Unit} \mid \alpha \mid \tau \to \tau \mid \forall\alpha.\,\tau \mid \mathsf{Ref}\ \tau\\ e ::= \mathtt{unit} \mid x \mid \lambda x{:}\tau.\,e \mid \Lambda\alpha.\,e \mid e_1\,e_2 \mid e\,[\tau]\\ \quad \mid \mathtt{new}\ e \mid \mathtt{deref}\ e \mid e_1 := e_2 \mid \mathtt{loc}\ l\\ v ::= \mathtt{unit} \mid \lambda x{:}\tau.\,e \mid \Lambda\alpha.\,v \mid \mathtt{loc}\ l\end{array}$$

Well formed types $\boxed{\Delta \models \tau}$

$$\Delta \models \mathsf{Unit} \qquad \Delta, \alpha \models \alpha \qquad \frac{\Delta \models \tau_1 \quad \Delta \models \tau_2}{\Delta \models \tau_1 \to \tau_2}$$

$$\frac{\Delta, \alpha \models \tau}{\Delta \models \forall\alpha.\,\tau} \qquad \frac{\Delta \models \tau}{\Delta \models \mathsf{Ref}\ \tau}$$

Typing $\boxed{\Gamma; \Delta \vdash e : \tau}$

$$\Gamma;\Delta;\mathrm{M} \vdash \mathtt{unit} : \mathsf{Unit} \qquad \Gamma, x:\tau;\Delta;\mathrm{M} \vdash x : \tau \qquad \frac{\Delta \models \tau \quad \Gamma, x:\tau;\Delta;\mathrm{M} \vdash e : \tau'}{\Gamma;\Delta;\mathrm{M} \vdash \lambda x{:}\tau.\,e : \tau \to \tau'} \qquad \frac{\Gamma;\Delta,\alpha;\mathrm{M} \vdash v : \tau}{\Gamma;\Delta;\mathrm{M} \vdash \Lambda\alpha.\,v : \forall\alpha.\,\tau}$$

$$\frac{\Gamma;\Delta;\mathrm{M} \vdash e_1 : \tau \to \tau' \quad \Gamma;\Delta;\mathrm{M} \vdash e_2 : \tau}{\Gamma;\Delta;\mathrm{M} \vdash e_1\,e_2 : \tau'} \qquad \frac{\Gamma;\Delta;\mathrm{M} \vdash e : \forall\alpha.\,\tau \quad \Delta \models \tau'}{\Gamma;\Delta;\mathrm{M} \vdash e\,[\tau'] : \tau\{\alpha \mapsto \tau'\}} \qquad \frac{\Gamma;\Delta;\mathrm{M} \vdash e : \tau}{\Gamma;\Delta;\mathrm{M} \vdash \mathtt{new}\ e : \mathsf{Ref}\ \tau}$$

$$\frac{\Gamma;\Delta;\mathrm{M} \vdash e : \mathsf{Ref}\ \tau}{\Gamma;\Delta;\mathrm{M} \vdash \mathtt{deref}\ e : \tau} \qquad \frac{\Gamma;\Delta;\mathrm{M} \vdash e_1 : \mathsf{Ref}\ \tau \quad \Gamma;\Delta;\mathrm{M} \vdash e_2 : \tau}{\Gamma;\Delta;\mathrm{M} \vdash e_1 := e_2 : \mathsf{Unit}} \qquad \Gamma;\Delta;\mathrm{M}, l:\tau \vdash \mathtt{loc}\ l : \mathsf{Ref}\ \tau$$

Semantics $\boxed{S; e \longrightarrow S'; e'}$

$$\frac{S; e_1 \longrightarrow S'; e_1'}{S; e_1\,e_2 \longrightarrow S'; e_1'\,e_2} \qquad \frac{S; e_2 \longrightarrow S'; e_2'}{S; v_1\,e_2 \longrightarrow S'; v_1\,e_2'} \qquad \frac{S; e \longrightarrow S'; e'}{S; e\,[\tau] \longrightarrow S'; e'\,[\tau]}$$

$$S; (\lambda x{:}\tau.\,e)\,v \longrightarrow S; e\{x \mapsto v\}$$

$$S; (\Lambda\alpha.\,v)\,[\tau] \longrightarrow S; v\{\alpha \mapsto \tau\}$$

$$S; \mathtt{new}\ v \longrightarrow S, l \mapsto v; \mathtt{loc}\ l$$

$$S, l \mapsto v; \mathtt{deref}\ (\mathtt{loc}\ l) \longrightarrow S, l \mapsto v; v$$

$$S, l \mapsto v'; \mathtt{loc}\ l := v \longrightarrow S, l \mapsto v; v$$

$$\frac{S; e \longrightarrow S'; e'}{S; \mathtt{new}\ e \longrightarrow S'; \mathtt{new}\ e'} \qquad \frac{S; e \longrightarrow S'; e'}{S; \mathtt{deref}\ e \longrightarrow S'; \mathtt{deref}\ e'}$$

$$\frac{S; e_1 \longrightarrow S'; e_1'}{S; e_1 := e_2 \longrightarrow S'; e_1' := e_2} \qquad \frac{S; e_2 \longrightarrow S'; e_2'}{S; v_1 := e_2 \longrightarrow S'; v_1 := e_2'}$$

Figure 1: The definition of $\lambda^{\forall,\mathrm{ref}}$.

```
types name = nat

datatype ty =
    TyUnit
  | TyBase name
  | TyFreeVar name
  | TyVar nat
  | TyArrow ty ty    (infixr → 900)
  | TyForall ty      (∀._ [900] 900)
  | TyRef ty
```

```
datatype tm =
    TmUnit
  | TmFreeVar name
  | TmVar nat
  | TmAbs ty tm      (λ[_]._ [900,800] 800)
  | TmApp tm tm      (infixl · 900)
  | TmProd tm        (Λ[*]._ [900] 900)
  | TmTapp tm ty     (_⟨_⟩ [900,0] 900)
  | TmNew tm
  | TmAssign tm tm   (infixr := 900)
  | TmDeref tm
  | TmLoc name
```

Figure 2: The syntax of $\lambda^{\forall,\mathrm{ref}}$ in Isabelle/HOL.

2 Syntax

The language $\lambda^{\forall,\mathrm{ref}}$ that we study is the polymorphic λ-calculus (F_2), extended with first-class ML-style references. No memory deallocation (`free`) is possible and there is a "value restriction" for polymorphic terms [13, p. 335–336]. The syntax, typing and call-by-value semantics of $\lambda^{\forall,\mathrm{ref}}$ is given in Fig. 1, in the way that it is commonly presented. In this and the following sections, we outline the definition of $\lambda^{\forall,\mathrm{ref}}$ in Isabelle/HOL and the proof of its type safety. We focus on explaining and justifying the design choices we made.

The syntax of $\lambda^{\forall,\mathrm{ref}}$, as encoded in Isabelle/HOL, is given in Fig. 2. In comparison to the standard syntax of Fig. 1, the first difference to notice is the addition of base types (which are easier to deal with, if we distinguish them from free type variables). The second, and most important difference is the use of a technique called "locally nameless" for dealing with bound variables [5].

In the locally nameless approach, named variables and DeBruijn indices [14] coexist in the representation of types and terms. Each abstraction (λ and Λ for terms, $\forall$ for types) introduces a new DeBruijn index, not a named variable. However, DeBruijn indices are replaced by named variables whenever we want to see *inside* terms. This means that any given term or type that we examine can-

not contain unbound DeBruijn indices (we call such terms and types *closed*.) In this way, we can define substitution without having to shift any DeBruijn indices. We define three kinds of substitution: `vsubst_ty` (types in types), `vsubst_tm` (terms in terms) and `vsubst_tmty` (types in terms). For example, `vsubst_ty` $\tau' \, i \, \tau$ stands for $\tau\{i \mapsto \tau'\}$. We also define `freshen_ty` $\alpha \, \tau$ as a shortcut for `vsubst_ty (TyFreeVar` α`)` $0 \, \tau$. Similarly for the other two kinds of substitution.

A definition that we found very useful is that of a *substitution function* for types. A substitution function captures the notion of a *context*, possibly containing holes in which closed types can be substituted. If f and g are two substitution functions, we can show that if $f(\alpha) = g(\alpha)$ for some fresh type variable α, then $f(\tau) = g(\tau)$ for all closed types τ.

3 Typing

Our encoding of the typing relation in Isabelle/HOL treats named variables and memory locations identically. In this way, we use the same environment for Γ and M (see Fig. 1). We chose to represent environments as sets. The type environment Δ contains type variables, while the term environment Γ contains pairs of term variables (or locations) and types.

The definition of *well formed* type ($\Delta \models \tau$) is almost straightforward: all named variables must exist in the given type environment Δ. The case of $\forall$-abstractions is a little tricky, as they introduce new DeBruijn indices in their bodies. For such an abstraction to be well formed, the body must be well formed when we substitute the DeBruijn index with a *fresh* named variable α, which is introduced in Δ. The notion of a *fresh* variable is also tricky. A fresh variable α must not appear in Δ or in the abstraction's body. However, this is not sufficient for proving essential properties of $\Delta \models \tau$, such as weakening (i.e. if $\Delta \models \tau$ and α is fresh, then $\Delta, \alpha \models \tau$). To bypass this problem, we also restrict α not to occur in an arbitrary finite set L, which intuitively represents "all other variables that should be avoided".

```
wf_forall[intro!] : ⟦ Δ ⊨ TY; finite L;
                      ∀a. ¬ Δ defines a ∧ ¬ a free in type τ ∧ a ∉ L ⟶
                        Δ,(a:⋆) ⊨ freshen_ty a τ ⟧ ⟹ Δ ⊨ ∀.τ
```

We must also restrict ourselves to *well formed* environments. Every finite Δ is well formed. For a Γ to be well formed, it must be finite, it must not attribute two different types to the same variable (or location) and all attributed types must be well formed. As Γ contains both named variables and locations, this definition imposes a name distinction between the two and is stricter than necessary.

```
Δ ⊨ TY  ≡  finite Δ
Γ;Δ ⊨ OK  ≡  finite Γ ∧ Δ ⊨ TY ∧ (∀x τ1 τ2. (x▷τ1)∈Γ ∧ (x▷τ2)∈Γ ⟶ τ1=τ2 ∧ Δ ⊨ τ1)
```

The typing judgement requires the substitution of DeBruijn indices with fresh named variables, in a way similar to what we used for well formed types. It is worth noting that no typing rule exists for DeBruijn indices and the typing rules for named variables and locations are almost identical. The following excerpt shows the typing rules for λ-abstractions, named variables and locations.

```
t_var[intro!] : ⟦ Γ;Δ ⊨ OK; (x▷τ) ∈ Γ ⟧ ⟹ Γ;Δ ⊢ TmFreeVar x : τ
t_abs[intro!] : ⟦ Γ;Δ ⊨ OK; finite L; Δ ⊨ τ1;
                  ∀x. ¬ x free in e ∧ x ∉ L ∧ ¬ Γ defines x ⟶
                    Γ,(x:τ1);Δ ⊢ freshen_tm x e : τ2 ⟧ ⟹
                  Γ;Δ ⊢ λ[τ1]. e : τ1→τ2
t_loc[intro!] : ⟦ Γ;Δ ⊨ OK; (l▷τ) ∈ Γ ⟧ ⟹ Γ;Δ ⊢ TmLoc l : TyRef τ
```

4 Semantics

We used a small step operational semantics for $\lambda^{\forall, \mathrm{ref}}$, similar to the one in Fig. 1. However, as we plan to extend $\lambda^{\forall, \mathrm{ref}}$ in the future with reference deallocation and will have to resort to a substructural type

system [15, ch. 1], we chose to store computed values in memory. Therefore, a store S contains both computed values and the contents of memory locations (references). It is a set of pairs of variables (locations) and terms. A store is *well formed* if it is finite, it does not bind a variable (location) to two different terms and all the terms it contains are indeed values.

```
S ⊨ Store  ≡  finite S ∧ (∀x v1 v2. (x↦v1)∈S ∧ (x↦v2)∈S ⟶ v1=v2 ∧ value v1)
```

Our semantics has an extra rule (`e_val`) that adds a computed value to a fresh variable (location) in the store. Thus, in our semantics, we treat variables (e.g. `TmFreeVar x`) in the same way that values are used in Fig. 1. To illustrate this, we give below our three evaluation rules that define (call-by-value) function application. It can easily be shown that the two semantics are equivalent.

```
e_val[intro!]    : ⟦ ¬ S defines z; S ⊨ Store; value v ⟧ ⟹ S;v ↪ S,(z↦v);TmFreeVar z
e_app_p1[intro!] : ⟦ S;e1 ↪ S';e1' ⟧ ⟹ S;e1·e2 ↪ S';e1'·e2
e_app_p2[intro!] : ⟦ S;e2 ↪ S';e2' ⟧ ⟹ S;(TmFreeVar x)·e2 ↪ S';(TmFreeVar x)·e2'
e_beta[intro!]   : ⟦ (z↦λ[τ].e1) ∈ S ⟧ ⟹ S;(TmFreeVar z)·(TmFreeVar y) ↪ S;freshen_tm y e1
```

5 Metatheory

As usual, we prove the type safety of $\lambda^{\forall,\mathrm{ref}}$ by proving two theorems: progress and preservation. Several standard lemmata are needed. Canonical form lemmata allow one to deduce the syntactic form of a well typed value, given its type. Weakening lemmata allow one to extend the environments in a judgement with fresh bindings. A substitution lemma, in general, states that typing is preserved when a term of the same type is substituted for a free variable. Given below are the definitions of our two main substitution lemmata, for terms and types. They are somewhat tricky, because of the locally nameless approach.

```
lemma substitution:
  assumes Γ;Δ ⊢ e' : τ'
  and Γ,(x:τ');Δ ⊢ vsubst_tm (TmFreeVar x) i e : τ
  and ¬ x free in e
  shows Γ;Δ ⊢ vsubst_tm e' i e : τ

lemma substitution_ty:
  assumes Δ ⊨ τ'
  and vsubst_ty_env (TyFreeVar a) i Γ;Δ,(a:⋆) ⊢
          vsubst_tmty (TyFreeVar a) i e : vsubst_ty (TyFreeVar a) i τ
  and ¬ a free in term e ∧ ¬ a free in type τ ∧ ¬ a free in env Γ ∧ ¬ Δ defines a
  and vsubst_ty_env τ' i Γ;Δ ⊨ OK
  shows vsubst_ty_env τ' i Γ;Δ ⊢ vsubst_tmty τ' i e : vsubst_ty τ' i τ
```

Both are proved by induction on the size of term e. The proof of the latter requires the property of substitution functions that was mentioned in §2.

The correspondence between a store S and typing environments Γ and Δ is captured by the *store typing* relation $\models S : \Gamma; \Delta$. The definitions of the two main theorems, preservation and progress, easily follow. Preservation states that operational semantics preserves typing, with possibly extended environments. Progress states that a well typed term is not *stuck*: either it is a variable (containing a computed value in the store), or the operational semantics can make one more evaluation step. Both theorems can be proved by induction on the typing derivation.

```
⊨ S : Γ;Δ  ≡ S  ⊨ Store ∧ (∀x τ. (x▷τ) ∈ Γ ⟶ (∃v. (x↦v) ∈ S ∧ Γ;Δ ⊢ v : τ))

theorem preservation:
  assumes Γ;Δ ⊢ e : τ
  and S;e ↪ S';e'
  and ⊨ S : Γ;Δ
  shows ∃Γ' Δ'. Γ ⊆ Γ' ∧ Δ ⊆ Δ' ⊨ S' : Γ';Δ' ∧ Γ';Δ' ⊢ e' : τ

theorem progress:
  assumes Γ;Δ ⊢ e : τ
  and ⊨ S : Γ;Δ
  shows not_stuck e S
```

6 Conclusions and Future Work

In this paper we outlined a mechanized proof of type safety for $\lambda^{\forall,\,\mathrm{ref}}$, the extension of F_2 with ML-style mutable references, using the Isabelle/HOL proof assistant. To the best of our knowledge, it is the first fully mechanized type safety proof for a language with polymorphic references in Isabelle/HOL. Additionally, it is the first fully mechanized type safety proof for a language with mutable references and impredicative polymorphism.

We intend to introduce an operator for explicit reference deallocation (`free`) in $\lambda^{\forall,\,\mathrm{ref}}$. To guarantee the type safety of the resulting language, we will employ a substructural type system. The interplay between linear pointer types (necessary for deallocation and strong assignment) and unrestricted pointer types (useful for dereferencing and weak assignment) is one of the primary interests of our future work.

References

[1] J.-Y. Girard, "Une extension de l'interprétation de Gödel à l'analyse, et son application á l'élimination des coupures dans l'analyse et la théorie des types," in *Proceedings of the 2nd Scandinavian Logic Symposium* (J. E. Fenstad, ed.), vol. 63 of *Studies in Logic and the Foundations of Mathematics*, pp. 63–92, North-Holland, 1971.

[2] J. C. Reynolds, "Towards a theory of type structure," in *Programming Symposium* (B. Robinet, ed.), vol. 19 of *Lecture Notes in Computer Science*, (Berlin), pp. 408–425, Springer-Verlag, 1974.

[3] M. Tofte, "Type inference for polymorphic references," *Information and Computation*, vol. 89, pp. 1–34, Nov. 1990.

[4] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer, 2002. Updated documentation available from `http://www.cl.cam.ac.uk/research/hvg/Isabelle/documentation.html`.

[5] C. McBride and J. McKinna, "Functional pearl: I am not a number; I am a free variable," in *Proceedings of the 2004 ACM SIGPLAN Haskell Workshop*, (New York), pp. 1–9, ACM Press, 2004.

[6] R. Harper, "A simplified account of polymorphic references," *Information Processing Letters*, vol. 51, no. 4, pp. 201–206, 1994.

[7] R. Harper, "A note on "A simplified account of polymorphic references"," *Information Processing Letters*, vol. 57, no. 1, pp. 15–16, 1996.

[8] B. Aydemir, A. Charguéraud, B. C. Pierce, and S. Weirich, "Engineering aspects of formal metatheory." Manuscript, available from `http://www.cis.upenn.edu/~bcpierce/papers/binders.pdf`, Apr. 2007.

[9] X. Leroy, "A locally nameless solution to the POPLmark challenge," Research report 6098, INRIA, Jan. 2007.

[10] D. K. Lee, K. Crary, and R. Harper, "Towards a mechanized metatheory of Standard ML," in *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pp. 173–184, 2007.

[11] C. Dubois, "Proving ML type soundness within Coq," in *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pp. 126–144, 2000.

[12] D. von Oheimb, *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. Available from `http://www4.in.tum.de/~oheimb/diss/`.

[13] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[14] N. G. de Bruijn, "Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation," *Indagationes Mathematicae*, vol. 34, pp. 381–392, 1972.

[15] B. C. Pierce, ed., *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.