Soundness of the Simply Typed Lambda Calculus in ACL2

Sol Swords
sswords@cs.utexas.edu

William R. Cook
wcook@cs.utexas.edu

Department of Computer Sciences
University of Texas at Austin

ABSTRACT

To make it practical to mechanize proofs in programming language metatheory, several capabilities are required of the theorem proving framework. One must be able to represent and efficiently reason about complex recursively-defined expressions, define arbitrary induction schemes including mutual inductions over several objects and inductions over derivations, and reason about variable bindings with minimal overhead. We introduce a method for performing these proofs in ACL2, including a macro which automates the process of defining functions and theorems to facilitate reasoning about recursive data types. To illustrate this method, we present a proof in ACL2 of the soundness of the simply typed λ-calculus.

Categories and Subject Descriptors

F.3.2 [**Theory of Computation**]: Semantics of Programming Languages

Keywords

Soundness, Lambda-Calculus, ACL2

General Terms

Theory languages

1. INTRODUCTION

Programming language metatheory, which includes the study of soundness of type systems and correctness of type-checking algorithms, is a tempting target for mechanized theorem proving. Desirable properties can usually be succinctly stated, but the complexity of the proofs grows with the number of syntactic constructs in the language. Often such proofs are strategically simple but can be long enough that they are tedious to write and difficult to check. Unfortunately, the details which make hand proofs tedious can also impede efforts to mechanize the proofs. A general challenge to mechanize programming language metatheory has been issued [1]. The challenge problems have already been solved using other theorem provers, including Coq [2] and Twelf [7]. Our goal in this work is to develop the infrastructure in ACL2 to eventually address these more challenging proofs.

In this paper we examine a proof in ACL2 of the soundness of the simply-typed λ-calculus, discussing problems which present themselves in going from the well-known hand proof to a mechanized proof. We present a helpful method of guiding proofs in this area. We also discuss a representation framework generated by a macro which facilitates reasoning about expressions of an abstract language syntax.

2. BACKGROUND

The abstract syntax of types and expressions for simply-typed λ-calculus with Booleans are as follows:

T	$::=$	$\mathsf{Bool} \mid T \to T$	*Types*
X, Y, Z	$::=$	$\mathsf{True} \mid \mathsf{False} \mid v$	*Simple Expressions*
		$\mid \lambda v{:}T \,.\, X$	*Abstraction*
		$\mid X\ Y$	*Application*
		$\mid \textbf{if}\ Z\ \textbf{then}\ X\ \textbf{else}\ Y$	*Conditionals*

Bool is the single base type, and the arrow type $A \to B$ is the type of a function which takes an argument of type A and produces a result of type B. Expressions consist of the Boolean constants True and False, variable references, λ-abstractions, function applications, and conditional expressions.

To formalize this description, the semantics of the language is defined by a standard small-step evaluation relation [8], a two-element relation between expressions. Elements of the relation are written $X \rightsquigarrow Y$, meaning X evaluates to Y in one step. Boolean constants and λ-abstractions are values, meaning they are considered to be fully evaluated.

The evaluation relation uses the capture-avoiding substitution $[v \mapsto Y]X$. In this form of substitution, any bound variables in X which are free in Y are renamed to fresh variable names before all free occurrences of v are replaced by Y. These rules then define the evaluation relation [8]:

$$\frac{X_1 \rightsquigarrow X_2}{X_1\ Y \rightsquigarrow X_2\ Y} \qquad \text{(E-APP1)}$$

$$\frac{X \text{ is a value} \qquad Y_1 \rightsquigarrow Y_2}{X\ Y_1 \rightsquigarrow X\ Y_2} \qquad \text{(E-APP2)}$$

ACL2 '06 Seattle, Washington USA

$$\frac{Y \text{ is a value}}{(\lambda v\!:\!T\,.\,X)\;Y \rightsquigarrow [v \mapsto Y]X} \qquad \text{(E-AppAbs)}$$

$$\frac{Z_1 \rightsquigarrow Z_2}{\textbf{if } Z_1 \textbf{ then } X \textbf{ else } Y \rightsquigarrow \textbf{if } Z_2 \textbf{ then } X \textbf{ else } Y} \qquad \text{(E-IfCond)}$$

$$\textbf{if } \mathsf{True} \textbf{ then } X \textbf{ else } Y \rightsquigarrow X \qquad \text{(E-IfTrue)}$$

$$\textbf{if } \mathsf{False} \textbf{ then } X \textbf{ else } Y \rightsquigarrow Y \qquad \text{(E-IfFalse)}$$

Requiring that X or Y be a value, although not absolutely necessary, has the effect of determining the order of evaluation of sub-expressions. Note that types are ignored by the evaluation relation.

The evaluation relation does not define transitions for values, because they are fully evaluated. But there are other expressions for which no transition is defined. Examples of such expressions are **if** $(\lambda v\!:\!\mathsf{Bool}\,.\,X)$ **then** True **else** False, in which a function appears where a boolean value is expected, and True False, which uses the boolean True where a function is expected. These expressions are syntactically well-formed but semantically nonsensical. They are examples of *type errors*, because of the mismatch between the type of value and the type of value that is needed. Such expressions are called *stuck*: they are not values but they cannot be evaluated further. The type system's purpose is to ensure that stuck expressions do not occur anywhere during evaluation.

The type system is based on a three-place relation, traditionally written as $\Gamma \vdash X : T$, which means "expression X has type T under typing context Γ." The typing context Γ (also called the type environment) is a list of assumptions of the types of variables, each written $v : T$, meaning variable v has type T. A term is considered well-typed if it has a type under the empty context, which is written as a blank, as in $\vdash X : T$.

The following rules define the typing relation [8]:

$$\Gamma \vdash \mathsf{True} : \mathsf{Bool} \qquad \text{(T-True)}$$

$$\Gamma \vdash \mathsf{False} : \mathsf{Bool} \qquad \text{(T-False)}$$

$$\frac{v : T \in \Gamma}{\Gamma \vdash v : T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, v : T_1 \vdash X : T_2}{\Gamma \vdash \lambda v\!:\!T_1\,.\,X : T_1 \rightarrow T_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash X : T_1 \rightarrow T \qquad \Gamma \vdash Y : T_1}{\Gamma \vdash X\;Y : T} \qquad \text{(T-App)}$$

$$\frac{\Gamma \vdash Z : \mathsf{Bool} \quad \Gamma \vdash X : T \quad \Gamma \vdash Y : T}{\Gamma \vdash \textbf{if } Z \textbf{ then } X \textbf{ else } Y : T} \qquad \text{(T-If)}$$

A type system is *sound* if repeated evaluation of a well-typed term never gives rise to a stuck expression (a non-value which has no possible evaluations). In order to state a soundess theorem in ACL2, an effective representation for terms is needed.

3. REPRESENTING TERMS AND TYPES

It is easy to design a simple representation in ACL2 of the λ-expressions and types using a list representation with distinguished symbols to denote different syntactic forms. However, in doing preliminary work on this problem we discovered that it could become very cumbersome to reason about such recursive data structures. In our case, a function which takes a λ-expression as input will have to distinguish between the six different syntactic forms and typically break down the components of the expression to construct new ones. Defining constructor and accessor functions helps in writing code, but if left enabled result in proofs which are cumbersome and difficult to read. Numerous trivial theorems are necessary in order to reason about these objects with function definitions disabled; with an incomplete set of these theorems most proofs quickly fail. We therefore created a macro which, given a description of the desired structure, defines all the functions and the theorems necessary to reason about them, allowing us to leave the function definitions disabled. After many revisions to the group of events submitted by this macro, we find that only in rare and specific cases is it necessary to re-enable the function definitions.

The macro we defined which generates these structures is named `defsum`, because we are defining a type which is a sum (or disjoint union) of several Cartesian products of types. Defsum differs from the `defstructure` book in ACL2 [3] in that it supports recursive typing quite naturally as well as mutually recursive structures, although it does not allow updating of components within a structure. The macro's syntax is a Lisp adaptation of the syntax for defining similar datatypes in languages such as ML and Haskell [5]. The abstract syntax of terms and types can be represented in a straightforward manner using `defsum`, as shown below.

```
(defsum stype
  (BOOL)
  (FUN (stype-p domain) (stype-p range)))
(defsum expression
    (TRUE)
    (FALSE)
    (LAM (varname-p var)
         (stype-p type)
         (expression-p body))
    (APP (expression-p fun)
         (expression-p arg))
    (VAR (varname-p name))
    (IFELSE (expression-p cond)
            (expression-p case1)
            (expression-p case2)))
```

As an aid to writing functions involving sum types, we have written a companion macro which performs pattern matching, binding variables to corresponding parts of an input term when the term is of the correct form. This macro, called pattern-match, is similar in function to case-match, but uses user-defined recognizers and accessors rather than operating directly on the list structure. Defsum produces the necessary events to allow pattern-match to recognize each product. For example, this function prints a representation of a type:

```
(defun print-stype (x)
  (pattern-match x
    ((BOOL) (cw "bool"))
    ((FUN a b) (cw "(~x0) -> (~x1)"
                   (print-stype a)
                   (print-stype b)))))
```

An additional difficulty impeding the adaptation of language metatheory to mechanical theorem proving technology is the representation of variable bindings. Hand proofs in this field usually ignore the technically difficult problems involved in formalizing these notions and instead assume that variable name conflicts never occur. When formalizing a language's semantics, however, it isn't possible to gloss over this issue. Several approaches for modeling variables and bindings have been developed. One approach is *higher-order abstract syntax* [6], in which binding constructs are represented by functions in the underlying language – this approach requires support for high-order functions, so it cannot be used in ACL2. Another popular strategy is to adopt a canonical form, such as de Bruijn notation [4], which replaces names with binding offsets. To follow the traditional presentation of syntax and typing rules as closely as possible, we have adopted the traditional approach in which variables are named and may be renamed in case of a conflict. This approach involves a manageable amount of work, namely defining the capture-avoiding substitution and proving a lemma about the effect of α-substitution on the typing relation.

4. PROGRESS AND PRESERVATION

The soundness of the simply-typed λ-calculus is stated as two theorems [8]. In combination they suffice to show that repeated evaluations of well-typed expressions can never result in an expression which is stuck in the sense that it has no evaluations but is not a value. The theorems are stated as follows.

Progress. If X is a well-typed expression, then either it is a value or there exists some expression X' such that $X \leadsto X'$.

Preservation. If X is a well-typed expression and $X \leadsto X'$, then X' is also a well-typed expression.

A "well-typed expression" here is X such that $\vdash X : T$ for some T; that is, the expression must have a type in the empty context. However, the preservation theorem as stated is too weak to prove by induction; we instead prove the theorem for an arbitrary context instead of just the empty context, and also require that the type of X' is the same as the type of X.

Preservation. If $\Gamma \vdash X : T$ and $X \leadsto X'$, then $\Gamma \vdash X' : T$.

In proving these properties, we found that it is extremely helpful to use an explicit representation for derivations of the typing and evaluation relations. The following defsum forms define their syntax.

```
(defsum type-deriv
  (T-TRUE)
  (T-FALSE)
  (T-VAR)
  (T-ABS (type-deriv-p body))
  (T-APP (stype-p argtype)
         (type-deriv-p fun)
         (type-deriv-p arg))
  (T-IF (type-deriv-p cond)
        (type-deriv-p case1)
        (type-deriv-p case2)))
(defsum eval-deriv
  (E-APPABS)
  (E-APP1 (eval-deriv-p fun))
  (E-APP2 (eval-deriv-p arg))
  (E-IFCOND (eval-deriv-p cond))
  (E-IFTRUE)
  (E-IFFALSE))
```

A `type-deriv` object represents a proof that a certain typing relation holds. Given a typing context, expression, type, and type derivation, we can recursively check that the type derivation corresponds to correct instantiations of the rules defining the typing relation. We define this operation in the function `valid-typing`. Similarly, we check for a valid evaluation relation in the function `valid-evaluation`, which takes two expressions and an evaluation derivation.

The progress and preservation theorems, stated in terms of typing and evaluation derivations, illuminate a new path toward proving them:

Progress. If there exists a valid derivation of $\vdash X : T$, then either X is a value or there exists an expression X' for which there is a valid derivation of $X \leadsto X'$.

Preservation. If there exist valid derivations of $\Gamma \vdash X : T$ and $X \leadsto X'$, then there exists a valid derivation of $\Gamma \vdash X' : T$.

The statements of these theorems make the method of proof clear. Given the objects mentioned in the hypotheses, we need to exhibit the objects postulated in the conclusions. We therefore define functions that produce the necessary objects. These functions are `next-expr` and `progress-deriv` which produce the new expression and evaluation derivation for progress, and `preservation-deriv` which produces the typing derivation for preservation. These are the statements of our progress and preservation theorems:

```
(defthm progress
  (implies
   (and (valid-typing nil expr type deriv)
        (not (value-p expr)))
   (valid-evaluation
    expr
    (next-expr expr type deriv)
    (progress-deriv expr type deriv))))
(defthm preservation
  (implies
   (and (valid-typing cntxt expr type type-deriv)
        (valid-evaluation expr expr2 eval-deriv))
   (valid-typing
    cntxt expr2 type
    (preservation-deriv
     cntxt expr type type-deriv eval-deriv))))
```

The progress theorem is simple to prove using the induction schema of the derivation-producing function, which recurses on the structure of the typing derivation. The proof of preservation is a straightforward induction on the evaluation derivation except in the E-AppAbs case, in which it must be shown that a substitution of an expression for a variable of the same type preserves the type of the outer expression. This substitution lemma is the largest part of the proof effort for the soundness theorems. It requires three sublemmas, listed below, which must be invoked in appropriate locations within its derivation building function. The textbook proof of preservation (as in [8], for example) uses all of these lemmas but the third, which is necessitated by our use of explicitly named variables.

Permutation. If $\Gamma \vdash X : T$ and Γ' assumes the same types as Γ for all variables appearing in Γ, then $\Gamma' \vdash X : T$.

```
(defthm permutation
  (implies
   (and (valid-typing cntxt1 expr type deriv)
        (env-same-bindings cntxt1 cntxt2))
   (valid-typing
    cntxt2 expr type
    (permutation-deriv cntxt1 deriv))))
```

Weakening. If $\Gamma, \Delta \vdash X : T$ and v does not appear in X, then $\Gamma, v : T_v, \Delta \vdash X : T$.

```
(defthm weakening
  (implies
   (and (valid-typing cntxt1 expr type deriv)
        (not (is-used-in var expr))
        (is-suffix suffix cntxt1)
        (equal cntxt2
               (insert-assoc
                var t2 suffix cntxt1)))
   (valid-typing
    cntxt2 expr type
    (weakening-deriv
     cntxt1 var t2 suffix deriv))))
```

Variable name substitution. If $\Gamma \vdash X : T$ and v_2 does not appear in X, then $[v_1 \mapsto v_2]\Gamma \vdash [v_1 \mapsto v_2]X : T$.

```
(defthm alpha-subst-ok
  (implies
   (and (valid-typing cntxt expr type deriv)
        (not (is-used-in var2 expr))
        (alpha-subst-env-okp
         var1 var2 suff cntxt)
        (equal cntxt2
               (env-subst-up-to
                var1 var2 suff cntxt)))
   (valid-typing
    cntxt2
    (alpha-subst var1 var2 expr)
    type
    (alpha-subst-deriv
     suff cntxt expr var1 var2 deriv))))
```

Substitution. If $\Gamma, v : T_v \vdash X : T$ and $\Gamma \vdash Y : T_v$, then $\Gamma \vdash [v \mapsto Y]X : T$.

```
(defthm substitution
  (implies
   (and (valid-typing cntxt val vtype vderiv)
        (valid-typing
         (cons (cons var vtype) cntxt)
         expr type deriv))
   (valid-typing
    cntxt
    (subst-expression val var expr)
    type
    (substitution-deriv
     cntxt var val vtype vderiv expr
     type deriv))))
```

For the first three of these lemmas, it is not necessary to construct a new derivation: in fact, the derivation function simply returns the derivation it is given, ignoring the other variables. In these cases it is still convenient to define a specific function to do this for each lemma. The rewrite rule resulting from each lemma operates only on terms which are calls of `valid-typing` on calls of the appropriate derivation function. Use of these derivation functions inside other derivation-producing functions causes the theorem prover to apply the corresponding lemma as a rewrite rule. This leads to a proof style that is similar to that of hand proofs: the user specifies the high-level strategy for each proof by defining the derivation function, and ACL2 grinds through the intuitive but tedious details which might be omitted in a hand proof.

Because these derivation functions are only used when we expect their corresponding lemmas to be applicable, we find it aids in both proof debugging and speed to force the hypotheses of such lemmas and set their backchain limits to zero. The hypotheses for each application of a lemma are then relieved in a separate forcing round, so that if there is a problem relieving one of the hypotheses, it is easy to see what lemma was being tried and why the proof failed.

5. CONCLUSIONS

Our method of reasoning about language metatheory consists of three major choices: first, a systematic and automated representation of terms, types, and derivations; second, explicit symbolic naming of variables; and third, the phrasing of lemmas as rewrite rules which are triggered by the presence of a particular function call, allowing us to explicitly and systematically guide the theorem prover. In future work, we hope to improve our treatment of variable naming in order to scale our method to more complex languages. One particular goal is to solve the PoplMark Challenge [1]. $F_{<:}$, the language used in the challenge, extends the λ-calculus with type abstractions and a subtyping relation. We have observed that these extensions significantly complicate reasoning about the naming of bound variables.

6. REFERENCES

[1] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Swewll, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *Proc. of the Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, 2005.

[2] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development.* Springer-Verlag, 2004.

[3] B. Brock. `defstructure` for ACL2 Version 2.0, 1997.

[4] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. In *Indagaciones Mathematische*, volume 34, pages 381–392, 1972.

[5] Haskell 98: A non-strict, purely functional language. Available at `http://haskell.org/onlinereport`, 1999.

[6] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 199–208, 1988.

[7] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In *Proc. of the Intl. Conf. on Automated Deduction (CADE-16)*, pages 202–206. Springer-Verlag LNAI 1632, 1999.

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