Theory Support for Weak Higher Order Abstract Syntax in Isabelle/HOL

Elsa L. Gunter
Department of Computer Science
University of Illinois
at Urbana - Champaign
egunter@illinois.edu

Christopher J. Osborn
Department of Computer Science
University of Illinois
at Urbana - Champaign
cosborn3@illinois.edu

Andrei Popescu [*]
Department of Computer Science
University of Illinois
at Urbana - Champaign
popescu2@illinois.edu

Abstract

We describe the theoretical underpinnings to support the construction of an extension to the Isabelle/HOL theorem prover to support the creation of datatypes for weak higher-order abstract syntax, and give an example of its application. This theoretical basis is centered around the concept of variable types (*i.e.* types whose elements are variables), and the concept of two terms in a given type having the "same structure" up to a given set of substitutions (the difference set) of one variable for another as allowed by that set. We provide an axiomatization of types for which the notion of having the same structure is well-behaved with the axiomatic class of *same_struct_thy*. We show that being a *same_struct_thy* is preserved by products, sums and certain function spaces.

Within a *same_struct_thy*, not all terms necessarily have the same structure as anything, including themselves. Those terms having the same structure as themselves relative to the empty difference set are said to be *proper*. A proper function from variables to terms corresponds to an abstraction of a variable in a term and also corresponds to substitution of variables for that variable in the term. Proper functions form the basis for a formalization of weak higher-order abstract syntax.

General Terms Verification, Languages

Keywords syntax with bindings, weak HOAS, proper functions, Isabelle/HOL, axiomatic classes

1. Introduction

For years there has been a desire to use the binding constructs of the logic of a theorem prover to represent the bound variables of abstract syntax of object languages. For example, when encoding syntax for first order logic with a type, say *fol*, we would like to be able to use functions into *fol* as the input to constructors for the "for all" and "exists" cases so that we would get α-conversion automatically. A major difficulty with using the function space *fol*$\Rightarrow$*fol* is that it is inconsistent to have a one-to-one constructor taking an argument of this type. One solution to this is instead to introduce a type of variables, say *var*, and to use the type *var*$\Rightarrow$*fol* for the arguments to constructors for binding constructs. This approach does allow for the constructors to be one-to-one. However, one difficulty with this approach is that it allows for more terms in the syntax (so-called *exotic* terms [4]) than should exist in the object language. For example, the term *exists*$(\lambda v.$ *if* $v = v_0$ *then true else false*$)$, where *true* and *false* are the obvious constants of type *fol* and v_0 is an arbitrary constant of type *var*, has no true meaning as syntax for a formula in first order logic.

The current version of Isabelle/HOL ([11]) supports a partial solution to this problem with the *Nominal* library [13, 16]. Using this library, one can create a *nominal_datatype* for creating the constructors for abstract syntax of languages with binders. When giving a *nominal_datatype*, one can specify that a variable is to be bound in the subsequent recursive argument. This gives us a "constructor" that is not one-to-one on its arguments, but rather is one-to-one on α-equivalence classes of input arguments. Then, when defining functions by primitive recursion, for the function to be well-defined, it is minimally necessary to show that such functions are invariant over the α-equivalence classes of the input to the binding constructors. In the current implementation, the user is required to prove that the operations passed to the primitive recursive operator use only finitely many free variables, and that they have correctly restricted the bound variables of the terms to be different from these free variables. While these restrictions may be desirable facts to know, requiring them for just the definition to be made poses a considerable extra burden for the user.

In this work, we describe a collection of mathematical theories, and a methodology for their use to enable the construction of weak higher order abstract syntax for object languages with binding constructs. In Section 2 we introduce a hierarchy of abstract theories, culminating in a theory of substitution of variables for free variables in a term. In Section 3, we discuss the possibility of a conservative extension to the existing *datatype* package to handle syntax with bindings, taking advantage of our axiomatic theory. In Section 4 we discuss the most closely related work. In Section 5 we outline some of the directions forward for this work. The scripts for our Isabelle theory described in this paper can be found at
http://www-faculty.cs.uiuc.edu/∼egunter/fms/proper

2. The Abstractions For Substitution

When defining a family of types for the abstract syntax of an object language with bound variables, one wishes to be able to reason about the syntax in a mostly first-order manner, but with the names of bound variables being "irrelevant," or always different from any collection of names that might be of immediate concern.

[*] Also: Institute of Mathematics Simion Stoilow of the Romanian Academy.

LFMTP '09 August 2, 2009, Montreal.
Copyright © 2009 ACM [to be supplied]. . . $10.00

That is, one expects to be able to take α-conversion for granted. A related concept also expected to be taken for granted is that of renaming free variables, or substituting variables for variables in terms. Therefore, we develop a theory of variables and their occurrence in terms, and a general theory of renaming free variables. These theories are realized using Isabelle's type classes.

Variables play a critical role in all the type classes developed in this work. To begin, therefore, we give a simple theory of variables, `var_thy`. Although this is the first axiomatic class to be created, it turns out, with appropriate additional definitions, to be an instance of every other abstract class we create. Following this, we introduce a class `free_vars_thy` for types whose elements contain finitely many free variables. Separately, we introduce the class `same_struct_thy` of types with the ability to say when two terms have the same structure up to substitutions of variables from a given set. The class `same_struct_free_vars_thy` is the merger of `free_vars_thy` and `same_struct_thy`. We extend `same_struct_free_vars_thy` by an assumption that the set of allowed substitutions need only involve the variables that actually occur free in the terms being compared to acquire `finite_same_struct_thy`. Independently, we extend `same_struct_thy` with the assertion that every term has the same structure as itself relative to the empty set of substitutions to arrive at `proper_thy`. The merge of `finite_same_struct_thy` and `proper_thy` is called `pre_subst_thy`. Finally, we extend `pre_subst_thy` with a notion of substitution of variables for free variables in a term to arrive at `subst_thy`. We describe each of these classes in greater detail below.

2.1 An Abstract Theory of Variables

The start of the axiomatic type classes we introduce to support weak higher-order abstract syntax in Isabelle/HOL is the axiomatic class of `var_thy`. In order to support the creation of types of syntax using a variety of different kinds of variables (*e.g.* syntax for polymorphic terms, with data variables and type variables), we need to be able to have multiple types of variables. These types of variables all have the same fundamental structure, but we need to keep them separate, hence the need for them to have different types. The fact that all types of variables have the same structure can be captured using Isabelle's notion of axiomatic classes [17] to restrict the types to those satisfying the properties we require of variables in general, via the axiomatic class `var_thy`.

Variables are a fundamental component of almost all formalisms of languages in computer science, and beyond. The main feature that is fundamental to standard uses of variables is the ability to pick a fresh variable, one distinct from a given finite set of variables. This is equivalent to requiring the set of all variables to be infinite. In our axiomatization of variables, this axiom is paramount.

An additional requirement typically placed on variables is that they form a *countable* set. That the set of all variables for a language be infinite is typically critical, since this is required to guarantee that it is always possible to pick a variable distinct from all those present in a given term. While countability is not usually strictly required, it is typically assumed to simplify issues such as making well-founded orderings on terms. In part because of this tradition, when formalizing an axiomatization of variables, we require any type representing variables to be countably infinite, with a bijection *var_to_nat*.

```
consts freshVar :: "'a set ⇒ 'a"
var_to_nat :: "'a ⇒ nat"
```

Since `var_to_nat` is to be a one-to-one correspondence we may define its inverse:

```
constdefs nat_to_var :: "nat ⇒ 'a"
"nat_to_var n ≡ SOME (v::'a). (var_to_nat::'a ⇒ nat) v
= n"
```

In our axiomatization of variables, we have chosen to maintain the requirement that the collection of all variables be countable for a second, more pragmatic reason: we wish to allow many types of variables in the datatypes we eventually wish to build. We wish these datatypes to satisfy certain properties of good behavior with respect to these variables. We further wish to capture this good behavior with Isabelle's axiomatic classes. However, axiomatic classes can have only one carrier type, represented as a type variable (the type of that class). It cannot express a relation between a collection of generic types, such as a type of variables and a type of expressions (terms). We will see an example of this difficulty in Subsection 2.1. We get around this problem by labeling each such concrete type with a unique identifier, and expressing these relations as relations with the combination of the type identifier and the projections of the variable types into the natural numbers.

```
types var_id = "string" — for using the name of a variable type as its identifier
consts var_id :: "'a ⇒ var_id"
types var_rep = "nat * var_id"
```

Since we have an infinite supply of strings, identifier/number combinations can capture any desired finite collection of variable types. Therefore, we have no loss of information, and at the same time, retain the convenience of using Isabelle type classes.

It does mean, however, that all our axioms for axiomatic classes that make use of variables must be rephrased in terms of the projections into the naturals. We state these axioms as "enumeration" axioms, and, where possible, derive the version stated directly in terms of variables from them. It is worth noting that the "enumeration" variants of the axioms are stronger, and often strictly so, because they state (an encoding of) facts about all variable types, while the non-enumeration variants can only refer to a single (polymorphic) type of variable.

We will be building a hierarchy of axiomatic classes relative to variables. There are two constants that are the basis of these classes: `free_vars_enum` – a function giving the free variables of a term, and `same_struct_enum` – a predicate saying when two terms differ by at most the substitutions in a given set. These constants are polymorphic on an Isabelle type variable `'a` representing a type of *terms* (or, more generally, of term-like items), as opposed to `'a` from the types of the previous constants (such as `var_to_nat`), which represents a type of *variables*. As indicated above, axioms may not directly mention any type variable other than the carrier of the class, and in particular axioms about terms can not directly mention types of variables. Therefore, the following constants are defined using natural numbers to stand in for variables. The constant `free_vars_enum` also has a version `free_vars` stated cleanly in terms of variables. The constant `free_vars` is defined in a uniform fashion for all types (of the appropriate classes) in terms of the constants defined using the naturals. On the other hand, both `free_vars_enum` and `same_struct_enum`, which are defined using the naturals, are overloaded, being defined on a per type basis. The types of the "enumerated" versions of the constants are as follows:

```
consts free_vars_enum :: "'a ⇒ var_rep set"
same_struct_enum :: "(var_id ⇒ (nat * nat) set) ⇒ 'a
⇒ 'a ⇒ bool"
```

For each class we define, we will wish to show that variables themselves are instances of that class. Since we wish totally uniform behavior of all types of variables, we add the definitions of these constants (that is, their "enumerated" versions) as additional axioms for the axiomatic class of variables. The axiomatization of variable types is then as follows:

```
axclass var_thy < type
freshVarNew_ax: "finite s ⟹ freshVar s ∉ s"
var_id_ax: "∀ (x::'a) (y::'a). (var_id x = var_id y)"
var_to_nat_inj_ax: "(var_to_nat(x::'a) = var_to_nat y) =
(x=y)"
var_to_nat_surj_ax: "∀ n. ∃ v. var_to_nat v = n"
var_free_vars_enum_def_ax: "free_vars_enum x={(var_to_nat
x,var_id x)}"
var_same_struct_enum_def_ax:
   "same_struct_enum vars x y =
    ((x = y) ∨ (var_to_nat x, var_to_nat y) ∈ vars
(var_id x))"
```

We characterize a type as a variable theory provided that the following hold:

- It is always possible to select a fresh variable distinct from any given finite set of variables.
- The type identifier is constant for the whole type.
- Variables are in one-to-one correspondence with the naturals.
- The free variables of that type in a variable are just that variable, while there are no free variables of any different type in it.
- Two variables have the same structure up to substitutions in a parametrized set `vars`, if the two variables are identical, or the pair is in the set `vars`.

With these axioms, we may show that `var_to_nat` and `nat_to_var` truly are inverses of each other.

With the existence of the axiomatic class `var_thy`, we are now in a position to give the definitions for the "non-enumerated" version of `free_vars`, which returns the set of free variables of that type.

```
constdefs free_vars :: "'a ⇒ ('b::var_thy) set"
"free_vars (s::'a) ≡ {(v::'b::var_thy). (var_to_nat v,
var_id v) ∈ free_vars_enum s}"
```

Having the above definition given uniformly in terms of the enumeration functions, we may now prove its expected definitions in the case of variables. For example, we have the following lemmas:

```
lemma var_free_vars_same_type:"free_vars (y::'a::var_thy)
= {y}"
```

The constant `same_struct_enum` can not be given a non-enumeration version. It is given a function parametrized by `var_id`'s, returning for each `var_id` a set of pairs of variables, which is the set of allowed replacements at that type. With such a parameterized set of substitutions, `same_struct_enum` then answers whether the next two given arguments are structurally the same up to replacing each variable in the first by a corresponding replacement variable given by the substitution set (or leaving it unreplaced). The constant `same_struct_enum` allows for the simultaneous substitution of variables for variables at multiple types in one term to acquire a second. Thus a non-enumerated version would require an extension to polymorphism beyond the limits of simple type theory.

We create one example of `var_thy` by making a copy of the naturals. In general, there will be as many distinct copies of this made as there are different types of variables in the user's expressions.

```
typedef var = "{x::nat. True}"
defs (overloaded)
"var_id (v::var) ≡ ''vars.var''"
"var_to_nat (v::var) ≡ Rep_var v"
"freshVar (s::var set) ≡ Abs_var(Suc(Max(Rep_var ‘ s)))"
```

The enumeration constants are defined as required by the theory axiomatization. With these definitions, we get that `var` is an instance of `var_thy`.

2.2 A (mini) theory of free variables

Now we begin to build our hierarchy of theories designed to represent abstractly types of syntax with bound variables. The first requirement we have for any such type is that it have a notion of the set of free variables in any term of that type, and that that set is always finite.

```
axclass free_vars_thy < type
  free_vars_finite_enum_ax:
"∀id::var_id. finite {v. v∈free_vars_enum e ∧ snd v =
id}"
```

It will be useful to know that a variety of types are in the class `free_vars_thy`, and a variety of type constructors preserve being in `free_vars_thy`. The types `unit` and `nat` are defined to have no occurrences of any variables of any type, and are therefore trivially in `free_vars_thy`.

```
defs (overloaded)
 "free_vars_enum (x::unit) ≡ {}"
 "free_vars_enum (x::nat) ≡ {}"
```

With these definitions, we may show that `unit` and `nat`, as well as every variable theory, are instances of `free_vars_thy`.

The free variables of products and sums are defined elementwise, using the notion of free variables from the underlying types.

```
primrec "sum_free_vars_enum (Inl x) = free_vars_enum x"
"sum_free_vars_enum (Inr y) = free_vars_enum y"

defs (overloaded) "free_vars_enum (x::('a+'b)) ≡
sum_free_vars_enum x"

primrec "prod_free_vars_enum (x,y)= (free_vars_enum x ∪
free_vars_enum y)"

defs (overloaded)
"free_vars_enum (x::('a*'b)) ≡ prod_free_vars_enum x"
```

We will include here the definition of `free_vars_enum` for functions with inputs of types belonging to the type class `var_thy`. However, we cannot in general prove that such function types are members of `free_vars_thy`.

```
defs (overloaded) "free_vars_enum f ≡
 {x. (∃ (y::'a::var_thy). x ∈ free_vars_enum (f y) ∧
(x≠ (var_to_nat y, var_id y)))}"

lemma var_fun_free_vars_def:
"free_vars f = {x. ∃ y. (x~= y) ∧ x ∈ free_vars (f y)}"
```

2.3 The theory `same_struct_thy`

The theory `same_struct_thy` axiomatizes the properties describing when two terms are the same (*i.e.* have the same structure) up to replacing zero or more occurrences of free variables by others as allowed by a given difference set. The substitutions performed allow for free occurrences of variables corresponding to the left naturals in pairs in the difference set to be replaced by the variables corresponding to the right naturals in the pairs. The axiomatic class axiomatizes the properties of the constant `same_struct_enum`, but to do so, it is assisted by a few auxiliary definitions: `set_flip` flips all the pairs in a set of pairs, and `set_trans_zip` composes two sets of pairs, but where we have to treat each set as if it contained the diagonal as well.

```
defs "insert_at i x S ≡ (λj. if i = j then insert x (S
j) else S j)"
 "delete_at i x S ≡ (λj. if i = j then (S j) - {x} else
S j)"
```

```
 "proj2_set S ≡ λ i. {x. (x,i) ∈ S}"
 "pair2_set S ≡ {(x,i). x ∈ (S i)}"
 "set_flip X ≡ λ i. {(y,x). (x,y)∈(X i)}"
 "set_trans_zip X Y ≡ λi. {(x,z) . (x,z) ∈ (X i) ∨
(x,z) ∈ (Y i) ∨ (∃ y. ((x,y) ∈ (X i) ∧ (y,z)∈ (Y
i)))}"
```

With these definitions, we axiomatize *same_struct_enum* as follows:

```
axclass same_struct_thy < type
 same_struct_enum_same_var_ax:
  "same_struct_enum (insert_at i (x,x) Vars) s t =
   same_struct_enum Vars s t"
 same_struct_enum_eq_ax:
  "same_struct_enum (λ i. {}) e1 e2 ⟹ (e1 = e2)"
 same_struct_enum_bigger_vars_ax:
  "⟦same_struct_enum Vars1 s t; (∀ i. Vars1 i ⊆ Vars2
i)⟧
  ⟹ same_struct_enum Vars2 s t"
 same_struct_enum_intersect_vars_ax:
  "⟦same_struct_enum Vars1 e1 e2; same_struct_enum Vars2
e1 e2⟧
   ⟹ same_struct_enum (λ i. Vars1 i ∩ Vars2 i) e1 e2"
 same_struct_enum_semi_com_ax :
  "same_struct_enum Vars a t ⟹ same_struct_enum
(set_flip Vars) t a"
 same_struct_enum_semi_trans_ax :
  "⟦same_struct_enum Vars1 e1 e2; same_struct_enum Vars2
e2 e3⟧
  ⟹ same_struct_enum (set_trans_zip Vars1 Vars2) e1
e3"
 same_struct_enum_free_vars_ax:
  "same_struct_enum Vars s t ⟹
   (∀ i. proj2_set (free_vars_enum s) i
    ⊆ (proj2_set (free_vars_enum t) i ∪ {x. ∃ y.
(x,y)∈Vars i}))"
 same_struct_enum_exists_ax: "∃ x. same_struct_enum (λ
i. {}) x x"
```

For any notion of two terms having the same structure under a given set of allowed variable substitutions, we require the following:

- Substituting a variable for itself does not affect whether two terms have the same structure.
- If two terms have the same structure with no substitutions, they must be equal.
- If two terms have the same structure under a given substitution, then they still do if we add in more allowed substitutions (they don't have to be used.)
- If two terms have the same structure under two different sets of allowed substitutions, then we only needed to use the substitutions common to both sets.
- If two terms have the same structure with respect to a set of substitutions, then they have the same structure the other way around with the inverse substitutions.
- If one term has the same structure as a second with respect to one set of substitutions, and the second term has the same structure as a third with respect to another set of substitutions, then the first has the same structure as the third with respect to the composition of the two sets of substitutions.
- If two terms have the same structure under a given substitution, then any free variable of the first is either free in the second or replaced by the substitution.
- There exists at least one term that has the same structure as itself under the empty substitution.

To begin to make use of the theory, we need to define *same_struct_enum* at various types. For *unit* and *nat* it is just defined to be equality. For *var* it is defined as the union of equality and the set of substitutions. For + and * it is defined elementwise. As with *free_vars_thy*, we show that *unit*, *nat*, and *var* are instances of *same_struct_thy*, and + and * preserve being a *same_struct_thy*.

As with *free_vars_enum*, we extend *same_struct_enum* to functions over variables as follows:

```
defs (overloaded) var_fun_same_struct_enum_def:
"same_struct_enum vars f1 f2 ≡
 ∀ v1 v2. same_struct_enum
   (insert_at (var_id v1)((var_to_nat v1),(var_to_nat
v2)) vars)
   (f1 v1) (f2 v2)"
```

With this definition, we may show that functions from variables to an instance of *same_struct_thy* again form an instance of *same_struct_thy*.

The amalgamation of *free_vars_thy* and *same_struct_thy* is *same_struct_free_vars_thy*.

```
axclass same_struct_free_vars_thy < free_vars_thy,
same_struct_thy
```

We augment *same_struct_free_vars_thy*, stating that we may restrict the substitution sets to variables that actually occur free in the terms concerned, to get *finite_same_struct_thy*.

```
axclass finite_same_struct_thy < same_struct_free_vars_thy
 same_struct_enum_free_vars_only_ax:
  "⟦same_struct_enum vars s t; ∀ i. finite (vars i) ⟧
⟹
   same_struct_enum
    (λi. (vars i ∩ ({x. (x,i)∈free_vars_enum s} ×
                    {x. (x,i)∈free_vars_enum t}))
     s t"
```

Again, we are able to show that *unit*, *nat*, and *var* are instances of *finite_same_struct_thy*, and + and * preserve being a *finite_same_struct_thy*.

Among the various terms of a type, we wish to pick out particularly well-behaved terms, those that have the "same structure" as themselves with respect to no substitutions. This concept is embodied in the term *proper*.

```
constdefs
proper :: "'a ⇒ bool"
proper_def: "proper t ≡ same_struct_enum (λ i. {}) t t"
```

To see the relevance of this concept, let us consider an example of a term that is not proper. The function $f = (\lambda v.\ \text{if } v = \mathsf{nat\_to\_var}\ 0 \text{ then } \mathsf{nat\_to\_var}\ 1 \text{ else } v)$ is not proper. For *f* to be proper $f(\mathsf{nat\_to\_var}\ v\ 0) = \mathsf{nat\_to\_var}\ v\ 1$ would have to have be the same as $f(\mathsf{nat\_to\_var}\ v\ 2) = \mathsf{nat\_to\_var}\ v\ 2$ up to replacing $\mathsf{nat\_to\_var}\ v\ 0$ by $\mathsf{nat\_to\_var}\ v\ 2$, which is not the case. Perhaps a simpler example is the function *var_to_nat*. No function that is sensitive to the name of the variable will be proper. For concrete syntax, the proper functions will correspond, as expected, to bindings of variables into terms. For instance, in (our representation of) the λ-calculus, we would have $\mathsf{Lam} :: (\mathit{var} \Rightarrow \mathit{term}) \Rightarrow \mathit{term}$ and $\mathsf{Lam} f$ would correspond to an actual λ-term if and only if f were proper.

Since, for some types, all their elements are proper, while for others, many are not, we introduce an axiomatic class to capture those with only proper elements.

```
axclass proper_thy < same_struct_thy
proper_ax: "proper v"
```

Once again, we are able to show that *unit*, *nat*, and *var* are instances of *proper_thy*, and + and * preserve being a *proper_thy*.

The axiomatic class *pre_subst_thy* is the union of *proper_thy* and *finite_same_struct_thy*.

```
axclass pre_subst_thy < proper_thy, finite_same_struct_thy
```

As we have seen, not all functions over variables are proper, but we are interested in those that are. To single them out, we introduce a new type *pfun* of proper functions from variable types to types having the notion of "same structure".

```
typedef ('a,'b) pfun =
 "{f::'a::var_thy ⇒ ('b::same_struct_thy). proper f}"
```

The definitions for *free_vars_enum* and *same_struct_enum* are lifted from those for functions.

```
defs (overloaded)
"free_vars_enum pf ≡ free_vars_enum (Rep_pfun pf)"
defs (overloaded) "same_struct_enum vars pf1 pf2 ≡
 same_struct_enum vars (Rep_pfun pf1) (Rep_pfun pf2)"
```

Where we could not show that the type of all functions over variables is a *free_vars_thy*, for the type of proper functions, we can. Likewise, we can show that it forms a *same_struct_thy*, a *proper_thy*, a *finite_same_struct_thy*, and hence a *pre_subst_thy*.

2.4 The theory *subst_thy*

The theory *subst_thy* extends the concept of having the same structure under a set of allowed substitutions with the concept of actually substituting one free variable with another. This can be done using the concept of *same_struct_enum* given in the previous subsection. So far, we have developed a lot of restrictions on the behavior of *same_struct_enum*, but none that force it to actually hold for any pair of terms except for pairs of the form an element paired with itself. However, for substitution to exist, we need to know that there exist pairs of terms having the same structure with one variable swapped for another. The single axiom of the theory *subst_thy* grants us just that.

```
axclass subst_thy < pre_subst_thy
same_struct_subst_exists_ax [rule_format]:
"∀ t x y i. x ≠ y ⟶
   (∃ t'. ((x,i)∉ free_vars_enum t') ∧
    same_struct_enum (insert_at i (x,y) (λ i. {})) t
t')"
```

It is relatively straightforard to show that *unit*, *nat* and every *var_thy* is a *subst_thy*, and that sums, products and *pfun* preserve being a *subst_thy*.

```
instance unit :: subst_thy
instance nat :: subst_thy
instance var_thy < subst_thy
instance "+" :: (subst_thy, subst_thy) subst_thy
instance "*" :: (subst_thy, subst_thy) subst_thy
instance "pfun" :: (var_thy,subst_thy)subst_thy
```

We actually require relatively little from the concept of a *subst_thy*, namely that a substitution actually exists. The reason we can require so little and yet prove interesting facts about this general concept of substitution is because of the rich theory already given for when two terms have the same structure modulo a set of allowed substitutions. From this system we derive a rich theory of substituting variables for variables.

The first thing to observe is that, because *pre_subst_thy* subsumes *finite_same_struct_thy*, the *t'* which the above axiom asserts exists, is in fact unique.

```
lemma same_struct_enum_unique:
"⟦x ≠ y ⟶ (x,i) ∉ free_vars_enum
(t'::'a::finite_same_struct_thy);
  same_struct_enum (insert_at i (x,y) (λ i. {})) t t';
  x ≠ y ⟶ (x,i) ∉ free_vars_enum t'';
  same_struct_enum (insert_at i (x,y) (λ i. {})) t t''⟧
⟹ t' = t''"
```

Knowing the uniqueness of such a *t'*, we are now in a position to define substitution of one variable for another in a term.

```
defs "subst_var y x t ≡ (THE t'. (x ≠ y ⟶ (var_to_nat
x, var_id x)∉ free_vars_enum t') ∧ (same_struct_enum
(insert_at (var_id x) (var_to_nat x, var_to_nat y) (λ i.
{})) t t'))"
```

From this definition of *subst_var* we may prove certain general expected behavior of *subst_var* with respect to *same_struct_enum* and *free_vars_enum*.

```
lemma subst_var_same_struct_enum:
"same_struct_enum
 (insert_at(var_id redex)(var_to_nat redex,var_to_nat
residue)(λi. {}))
 (s::'a::subst_thy) (subst_var residue (redex::'b::var_thy)
s)"
```

```
lemma var_subst_var_enum_def:
"(∀ (redex::'a::var_thy) residue (x::'a).
     subst_var residue redex x = (if redex = x then
residue else x)) ∧
 (∀ (redex::'a::var_thy) residue (x::'b::var_thy).
    ((var_id redex) ≠ (var_id x)) ⟶ (subst_var
residue redex x = x))"
```

At the types *nat* and *unit*, *subst_var* perform no change on the term given since by definition no variables actually occur in them.

```
lemma unit_subst_var_def: "subst_var
(residue::'a::var_thy) redex (s::unit) = s"
```

```
lemma nat_subst_var_def: "subst_var
(residue::'a::var_thy) redex (s::nat) = s"
```

For + and *, we have that *subst_var* behaves elementwise:

```
lemma sum_subst_var_inl:
 "subst_var residue(redex::'c::var_thy)((Inl
s)::(('a::subst_thy)+('b::subst_thy)))
  = Inl (subst_var residue redex s)"
```

```
lemma sum_subst_inr:
 "subst_var residue(redex::'c::var_thy)((Inr
t)::(('a::subst_thy)+('b::subst_thy)))
   = Inr (subst_var residue redex t)"
```

```
lemma prod_subst_var:
 "subst_var residue (redex::'c::var_thy)
(s::('a::subst_thy),t::('b::subst_thy))
  = (subst_var residue redex s, subst_var residue redex
t)"
```

An important characterization of proper functions may be had in terms of substitutions - namely, that a function from variables to a *subst_thy* type is proper if and only if it is a renaming of a variable in a term:

lemma *subst_var_pfun:*
"proper (f::('a::var_thy ⇒ 'b::subst_thy)) =
(∃ t x. f = (λy. subst_var y x t))"

Beyond theorems specific to specific types and type constructors, we develop a rich theory of the general behavior of substitutions. Additional theorems we prove about substitution include:

lemma *subst_var_commute:*
"⟦(redex1::'a::var_thy)≠redex2; residue1≠redex2;
residue2≠redex1⟧ ⟹
subst_var residue1 redex1 (subst_var residue2 redex2
(e::'b::subst_thy))
  = subst_var residue2 redex2 (subst_var residue1 redex1
e)"

lemma *subst_var_compose:*
"subst_var (residue::'a::var_thy) residue1 (subst_var
residue1 redex e) =
 subst_var residue residue1 (subst_var residue redex
(e::'b::subst_thy))"

lemma *subst_var_id: "subst_var (x::'a::var_thy) x s =*
(s::'b::subst_thy)"

lemma *subst_var_vacuous[rule_format]:*
"(redex::'a::var_thy) ∉ (free_vars::'b ⇒ 'a set) e ⟶
 subst_var residue redex e = (e::('b::subst_thy))"

lemma *residue_in_subst:*
"redex ∈ (free_vars::'b ⇒ 'a set) e ⟶
(residue::'a::var_thy) ∈
  free_vars (subst_var residue redex (e::'b::subst_thy))"

lemma *redex_not_in_subst:*
"∀ redex residue.
(redex ≠ residue ⟶ (redex::'a::var_thy)
 ∉ (free_vars::'b ⇒ 'a set) (subst_var residue redex
(e::'b::subst_thy)))"

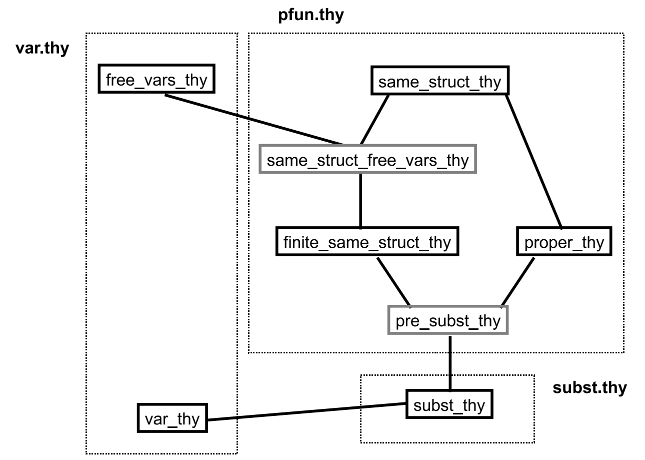

Figure 1. The class hierarchy

3. A planned extension to `datatype`

To render the theories of the previous section practically useful, it is desirable to have the ability to easily create new concrete types that are automatically shown be of sort *subst_thy*. Because we wish to use such types to model the abstract syntax of programming language constructs, we would like these types to be specified as free algebras over given sets of constructors, *i.e.* given by *datatype* specifications.

The current *datatype* package can handle, as the types allowed as arguments to the constructors, types built recursively from previously existing types and type variables; simple recursive, mutual recursive, and nested recursive occurrences of the types being defined; and function types where the type(s) being defined occur (possibly inside existing datatype constructors) only strictly positively.

This setting is not as flexible as we desire because we would effectively like to employ not only full function spaces, but also proper function spaces. Luckily, it turns out that all of the theory (induction, case analysis, primitive recursion, *etc.*) is equally valid in any case where function spaces are replaced by types injectable into the function spaces. In fact, the results obtained in the more general case may be readily derived from the corresponding definition in the existing *datatype* framework by composing the various constructors and operators (such as the case function and the primitive recursion operator) with the given injection and any corresponding projection function.

This demonstrates the legitimacy and possibility of extending the *datatype* package with type constructors injectable into the function space. As an example of what such an extension should have to offer, let us consider the following presumptive user specification of Church-style System F:[1]

```
var_types dvar tvar

proper_datatype
  dterm = inDvar dvar
          AppD "dterm * dterm"
          AppT "dterm * tterm"
          LamD "tterm * (dvar,dterm)pfun"
          LamT "(tvar,dterm)pfun"
and
  tterm = inTvar tvar
          Arr "tterm * tterm"
          Prod "(tvar,tterm)pfun"
```

After the indication of the two variable types, dvar and tvar, the user gives a "proper_datatype" declaration, defining the types of dterm and tterm together as fixed point for some equations involving in the righthand sides sums, products and the proper function space (pfun). This resembles closely the set-theoretic least fixed point mechanism underlying Isabelle's standard datatype facility; note however that the notion of a proper function is technically only available (via a recursively defined same_struct relation) *a posteriori*, after terms have already been defined. While clearly the intended types dterm and tterm do form a fixed point for the above equations, it is not clear to us at this moment with respect to which structure they form the *least* fixed point.[2]

For now, we can implement dterm and tterm using the roundabout route of first defining the types of *quasi-terms* as a regular Isabelle datatype which employs arbitrary (i.e., non-necessarily proper) function spaces. First, we define the two types of variables as countable types:

[1] Below, in the names of the Isabelle types "d" as a prefix stands for "data" and "t" for "type", and similarly in the names of the operators "D" as the suffix stands for "Data" and "T" for "Type". Thus, we have the types of data variables and terms and of type variables and terms, and constructors for data and type application and abstraction. Arr is the constructor for arrow (i.e., object-level function) types.

[2] Previous work on recursion principles specific to syntax with bindings, such as [5, 8, 15] could provide an elegant answer to this – we plan to investigate this in a future paper.

```
typedef dvar = "{x::nat. True}"
typedef tvar = "{x::nat. True}"
```

then define same_struct_enum and free_vars_enum for them as expected and prove these two types to be instances of var_thy. Next, we define the datatypes Qdterm and Qtterm, of quasi-terms:

```
datatype
  Qdterm = QinDvar dvar
           QAppD "Qdterm * Qdterm"
           QAppT "Qdterm * Qtterm"
           QLamD "Qtterm * (dvar => Qdterm)"
           QLamT "tvar => Qdterm"
and
  Qtterm = QinTvar tvar | QArr "Qtterm * Qtterm"
           QProd "tvar => Qtterm"
```

same_struct_enum and free_vars_enum are then defined for quasi-terms. E.g., the definition of same_struct_enum for data terms goes as follows (the definition for type terms is omitted):

```
"same_struct_enum Vars Z Z' =
(case (Z,Z') of
   (QinDvar x, QinDvar x') =>
    same_struct_enum Vars x x'
  |(QAppD(X,Y), QAppD(X',Y')) =>
    same_struct_enum Vars X X' /\
    same_struct_enum Vars Y Y'
  |(QAppT(X,A), QAppT(X',A')) =>
    same_struct_enum Vars X X' /\
    same_struct_enum Vars A A'
  |(QLamD(A,XX), QLamD(A',XX')) =>
    (ALL x x'. same_struct_enum (insert_at ''dvar''
    (var_to_nat x, var_to_nat x') Vars) (XX x)
    (XX' x')) /\ same_struct_enum Vars A A'
  |(QLamT AX, QLamT AX') =>
    (ALL a a'. same_struct_enum (insert_at ''tvar''
    (var_to_nat a, var_to_nat a') Vars) (AX a)
    (AX' a'))
  |_ => False)"
```

Note that the package would use the Isabelle strings "dvar" and "tvar" for representing the var_id's of the two types of variables.

Next, the desired types of terms are defined via the predicate proper (a priori associated to same_struct_enum for arbitrary types):

```
typedef dterm = "{X::Qdterm. proper X}"
typedef tterm = "{A::Qtterm. proper A}"
```

Next, using the back-and-forth representation and abstraction maps between the types of quasi-items and those of "true" items, we define same_struct_enum and free_vars_enum, as well as the syntactic constructs, for "true" items, of types dterm and tterm. E.g., the syntactic constructors are defined as follows:

```
"inDvar x = Abs_dterm (QinDvar x)"
"AppD(X,Y) = Abs_dterm(QAppD(Rep_dterm X,
                             Rep_dterm Y))"
"AppT(X,A) = Abs_dterm(QAppT(Rep_dterm X,
                             Rep_tterm A))"
"LamD(A,XX) = Abs_dterm(QLamD(Rep_tterm A,
%x. Rep_dterm(Rep_pfun XX x)))"
"LamT AX = Abs_dterm(QLamT(%a. Rep_dterm (Rep_pfun
AX a)))"

"inTvar a = Abs_tterm (QinTvar a)"
"Arr(A,B) = Abs_tterm(QArr(Rep_tterm A, Rep_tterm B))"
"Prod AA = Abs_tterm(QProd(%a. Rep_tterm (Rep_pfun
AA a)))"
```

Then the types dterm and tterm are proven to be instances of subst_thy, which rewards us with a definition of substitution and all the theorems in Section 2. Additionally, injectiveness and disjointness of the syntactic constructs follow at once from the corresponding properties for quasi-items:

```
theorem inj:
"(inDvar x = inDvar y) = (x = y) /\
 (inTvar a = inTvar b) = (a = b) /\
 (AppD(X,Y) = AppD(X',Y')) = (X = X' /\ Y = Y') /\
 (AppT(X,A) = AppT(X',A')) = (X = X' /\ A = A') /\
 (LamD(A,XX) = LamD(A',XX')) = (A = A' /\ XX = XX') /\
 (LamT AX = LamT AX') = (AX = AX') /\
 (Arr(A,B) = Arr(A',B')) = (A = A' /\ B = B') /\
 (Prod AA = Prod AA') = (AA = AA')"

theorem diff:
"inDvar x ~= AppD(X,Y) /\ inDvar x ~= AppT(X,A) /\
 inDvar x ~= LamD(XX) /\ inDvar x ~= LamT(AX) /\
 AppD(X,Y) ~= AppT(X,A) /\  AppD(X,Y) ~= LamD(XX) /\
 AppD(X,Y) ~= LamT(AX) /\ LamD(XX) ~= LamT(AX) /\
 inTvar a ~= Arr(A,B) /\ inTvar a ~= Prod AA /\
 Arr(A,A') ~= Prod AA"
```

Induction and recursion principles also follow immediately from corresponding principles on quasi-items. For example, an iterator would merely "copy" the built-in iterator for the standard datatype of quasi-items, and would have the type:

```
proper_iter :: "(dvar => 'a) =>
                ('a * 'a => 'a) =>
                ('a * 'b => 'a) =>
                ('b * (dvar => 'a) => 'a) =>
                ((tvar => 'a) => 'a) =>
                (tvar => 'b) =>
                (tvar * tvar => tvar) =>
                ((tvar => 'b) => 'b) =>
                (dterm => 'a) * (tterm => 'b)"
```

The nominal package [16] provides an alternative to this approach, but in that case, the solution provided is not a free algebra, but one where a predetermined set of equations must hold. These equations effectively cause the constructors to be one-to-one on congruence classes (α-equivalence classes) rather than one-to-one on their arguments. In our setting, the constructors are one-to-one on their arguments, with no extra equations imposed. We believe this will simplify the task of defining functions and relations over the types to be constructed. That being said, there is an isomorphism between the types defined in the nominal logic and the equivalent ones given in our proposed extension.

4. Related Work

There has been much work performed in the area of higher-order abstract syntax. We will discuss only that which seems most closely related to the work in this paper. In particular, we shall not discuss *axiomatic* approaches for representing syntax with bindings (such as [9, 7, 12]), [3] but only frameworks which are built, like ours, as definitional packages in a general-purpose theorem prover.

The Isabelle/HOL Nominal package [16] is based on the Nominal Logic [13] achieves a similar result to that of the package we are developing. The theory behind this work is that of α-equivalence,

[3] Note, however, that the axiomatic setting in [9] has been proven consistent via sheaf-theoretic models in [3, 6].

and its equational rendering via the *swap* function, and the constructors created are not one-to-one on their arguments, but rather are one-to-one on α-equivalence classes. This means that whenever a function is defined by structural recursion, it is necessary to show that the function is well-defined, *i.e.*, it is constant on α-equivalence classes. In our work, since we use functions instead of α-equivalence classes, our constructors are one-to-one directly for their arguments. Thus functions defined by primitive recursion are automatically well-defined. However, in some cases, proof obligations involving properness may arise.

Another difference between our work and that of the Nominal package is that in our framework it will be possible to add new variable types incrementally, allowing previously existing theories with a variety of variable types to be imported into new theories needing to create new variable types. Our mechanism does expect that each newly created variable type will be given an identifier that is distinct from all previously existing variable types. In our scheme, the full name, including the theory name, is used. These are clearly all distinct. Should a user choose to create, by hand so to speak, a new variable type having the same identifier as a previously existing type, and were to then combine these two types of variables when creating a new *subst_var_thy* type, no inconsistency will occur. Nevertheless, when reasoning about these terms, particularly when substitution is involved, odd behavior will occur. When a substitution of a variable of one type occurs, it may result in another variable of the other type also being changed. Other problems will include the set of free variables being larger than one would expect. These problems are a result of our attempt to push the type system of Isabelle/HOL a bit past its limits, but can easily be avoided by creating variable types in the intended manner.

In [4], weak higher-order abstract syntax is used in the Calculus of Inductive Constructions for encoding the syntax of object languages with bindings. To avoid the occurrence of "exotic" terms, they introduce a predicate "valid_0" to eliminate terms that do not represent elements of the object language. This notion is closely related to that of *proper*. (*proper* easily implies valid_0 for the datatype *lam*, but the other way around is a bit harder to show.) Similar work has resulted in a shallow embedding of the π-calculus in Isabelle/HOL in [14], also employing insight from the Theory of Contexts[9]. These works do not build frameworks for the automatic creation of weak higher-order abstract syntax datatypes in general (although they hint at the possibility.) Moreover, we provide a rich theory of substitution of variables for variables, preproven in HOL, using axiomatic classes.

Hybrid [1] is a system formalized originally in Isabelle and later also in Coq [10], designed with a goal similar to ours. It employs the untyped λ-calculus with constants, implemented with de Bruijn indexes and featuring higher-order operators defined on top of the implementation, in such a way that λ-calculus bindings are captured by Isabelle functional bindings (thus integrating the λ-terms shallowly into the Isabelle meta-layer). Their approach falls within *strong HOAS*, since λ-abstractions are modeled using functions from terms to terms. As customary in HOAS-like representations in general-purpose frameworks, one also needs to single out *proper* functions there, i.e., those functions from terms to terms which do represent actual λ-calculus terms. Thanks to their shallow (a posteriori) representation, this λ-calculus syntax accommodates specification of arbitrary syntax with static bindings via a standard HOAS encoding. However, this style of encoding is not, in our opinion, as convenient as having the desired syntax with bindings available in a direct fashion, like the Nominal package is doing and like our planned package will do. Concerning the topic of encoding inference (which we have not touched in this paper), the latest version of Hybrid takes a three-level architecture approach, similar to the one employed in genuinely HOAS frameworks such as Twelf [12] and Abella [7].

5. Future Directions

Obviously, the immediate future direction is to complete the package to automatically generate proper datatypes. Such a package will not only generate definitions and theorems comparable to those created by the current datatype package, but also will define the two notions of *free_vars_enum* and *same_struct_enum*, and prove that the defined datatype is a *subst_thy*, provided that all auxiliary types are also *subst_thy*s. It will prove the recursive "definition" of *subst_var*. It will also define versions of the constructors lifted to create proper functions for the type being defined. In conjunction with this package, we will provide a package for automatically creating new types of variables.

Beyond this basic support, it should also be possible to represent the proper functions as an algebraic datatype with constructors and a principle of primitive recursion in its own right. The constructors for such a representation would be lifted versions of the constructors for the original datatype. Each original constructor taking variables as arguments would generate two lifted constructors per variable. Such a representation would allow the creation of proper functions without the need to resort to first principles to prove they were proper. Further it seems possible that combinators for proper functions in general could be defined.

To demonstrate the utility of this package we will implement test examples such as the $F_{<}$, as suggested in [2].

References

[1] Simon Ambler, Roy L. Crole, and Alberto Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In *TPHOLs '02: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, pages 13–30, 2002.

[2] 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 *Theorem Proving in Higher Order Logics, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, August 2005.

[3] Anna Bucalo, Furio Honsell, Marino Miculan, Ivan Scagnetto, and Martin Hofmann. Consistency of the theory of contexts. *J. Funct. Program.*, 16(3):327–372, 2006.

[4] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 902 of *LNCS*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag.

[5] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding (extended abstract). In *Proc. 14$^{\text{th}}$ LICS Conf.*, pages 193–202. IEEE, Computer Society Press, 1999.

[6] Murdoch James Gabbay and Martin Hofmann. Nominal renaming sets. In *LPAR*, pages 158–173, 2008.

[7] Andrew Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of IJCAR 2008*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161. Springer, August 2008.

[8] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, page 204, 1999.

[9] Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *Leeuwen*

(Eds.), 28th International Colloquium on Automata, Languages and Programming, ICALP 2001, pages 963–978, 2001.

[10] Alberto Momigliano, Alan J. Martin, and Amy P. Felty. Two-level hybrid: A system for reasoning using higher-order abstract syntax. *Electron. Notes Theor. Comput. Sci.*, 196:85–93, 2008.

[11] T. Nipkow, L. C. Paulson, , and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2002.

[12] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16*, pages 202–206. Springer-Verlag LNAI, 1999.

[13] Andrew M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006.

[14] Christine Röckl and Daniel Hirschkoff. A fully adequate shallow embedding of the [pi]-calculus in Isabelle/HOL with mechanized syntax analysis. *J. Funct. Program.*, 13(2):415–451, 2003.

[15] Yong Sun. An algebraic generalization of Frege structures - binding algebras. *Theor. Comput. Sci.*, 211(1-2):189–232, 1999.

[16] Christian Urban, Julien Narboux, and Stefan Berghofer. *The Nominal Datatype Package*, 2007.

[17] M Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics, TPHOLs'97*, volume 1275 of *LNCS*, pages 307 – 322, Murray Hill, NJ, USA, August 1997. Springer.