A Simple and General Theoretical Account for Abstract Types

Hongwei Xi

Boston University

Work partly funded by NSF grants no. CCR-0229480 and no. CCF-0702665

Program Organization is Vital

- In general, software evolves constantly in order to accommodate emerging needs that are often difficult to foresee
- The ability to effectively localize changes made to existing programs is of great importance during software development, maintainance and evolution

A Fundamental Problem

- As experience indicates, a fundamental problem in the design of a module system lies in properly addressing the tension between the need for hiding information about a program unit from others and the need for propogating information between program units.
- The former need helps the construction of a program unit in relative isolation
- The latter need helps the assembly of program units into a coherent whole.

Abstract Types

A common approach to hiding implementation details is through the use of abstract types. For instance, the following presents a simple example of abstract type:

```
abstype intset // abstract type for integer sets

extern fun singleton (x: int): intset
extern fun member (xs: intset, x: int): bool
extern fun union (xs: intset, ys: intset): intset
extern fun intersect (xs: intset, ys: intset): intset
```

Note that this is a functional design (in contrast to an object-oriented design).

Standard Justification for Abstract Types

Abstract types are existential type (Mitchell and Plotkin). In terms of the *intset* example, we have something like an existentially quantified record type given as follows:

```
\exists X. {
  singleton = int -> X
, member    = (X, int) -> bool
, union     = (X, X) -> X
, intersect = (X, X) -> X
} // end of the record
```

Standard Justification Is Inadequate

It is well-known from the very beginning that treating abstract types as existential type is inadequate:

- Existential quantification often hides too much information. Even the names of abstract types are completely lost due to existential quantification.
- This can cause some serious problems when sharing of program units is of the concern. For instance,
  - Lexer makes use of a symbol table
  - Parser makes use of a symbol table
  - How can it be enforced that Lexer and Parser make use of the *same* symbol table implementation?

Conditional Type Equality

Let τ range over types. A judgment for conditional type equality is of the following form:

$$\phi \vdash \tau_1 = \tau_2$$

where ϕ is a set of type equality assumptions: $\tau_i^1 = \tau_i^2$ for $i = 1, \ldots, n$. For instance,

$$intset = list(int) \quad \vdash \quad (int \to intset) = (int \to list(int))$$

$$intset = list(int) \quad \vdash \quad ((intset, intset) \to intset) = ((list(int), list(int)) \to list(int))$$

In the case where ϕ is empty, the type equality becomes unconditional. Note that the general form of conditional type equality is undecidable.

Bindings for Type Constructors

A binding for a type constructor TC is of the following form

$$TC(\alpha_1, \ldots, \alpha_n) = \tau$$

where $\alpha_1, \ldots, \alpha_n$ are distinct type variables.

In this paper, for each conditional type equality judgment $\phi \vdash \tau_1 = \tau_2$, only bindings for type constructors are allowed to occur in ϕ.

A Simple Example

```
local

assume intset = list int

in

implement singleton (x) =
  list_cons (x, list_nil)

implement member (xs, x) = case+ xs of
  | list_cons (x1, xs1) =>
      if x = x1 then true else member (xs1, x)
  | list_nil () => false
// end of [member]

(* other functions are omitted *)

end // end of [local]
```

Another Simple Example

```
// an abstract type constructor declared somewhere
abstract set (a: type) // type for polymorphic sets
fun singleton {a:type} (x: a): set a
fun member {a:type}
  (xs: set a, x: a, eq: (a, a) -> bool): bool
fun union {a:type}
  (xs: set a, ys: set a, eq: (a, a) -> bool): set a
fun intersect {a:type}
  (xs: set a, ys: set a, eq: (a, a) -> bool): set a

// an implementation of polymorphic sets elsewhere

local

assume set (a) = list (a)

in

(* some code for various functions on sets *)

end // end of [local]
```

A typed language $\lambda^{\supset}$

A language $\lambda^{\supset}$ is formed in the paper, where a typing judgment is of the following form

$$\vec{\alpha}; \Gamma \vdash_{\phi} e : \tau$$

Some restrictions need to be imposed on ϕ in order to establish the type soundness for $\lambda^{\supset}$.

Linearity

Obviously, type soundness can be destroyed immediately if an abstract type is allowed to be implemented twice. So each abstract type is only allowed to be implemented once.

Linearity Enforcement

Linearity can be readily enforced in practice. For instance,

- for each implementation of an abstract type, we may insert a dynamic check to verify at *run-time* whether the abstact type has already been implemented elsewhere; or
- we may verify that no abstract type is implemented repeatedly at *link-time* by associating a global variable with each abstract type.

Cycles in Bindings for Type Constructors

There may be cycles in bindings for type constructors. For instance, we may have something like:

```
local

assume TC1 = TC2 -> TC3
assume TC2 = TC1; assume TC3 = TC1

in

...

end
```

Such cycles do not affect type soundness, but they can greatly complicate (practical) type checking. We currently disallow cycles.

By the way, ...

By the way, there are no cycles in the following code:

```
local
assume TC1 = TC2 -> TC3
in
...
end
(* ****** ****** *)
local
assume TC2 = TC1; assume TC3 = TC1
in
...
end
```

Splitting Recursive Type Declarations (1)

```
datatype boolexp =
   | Bool of bool
   | IntEq of (intexp, intexp)
// end of [boolexp]

and intexp =
   | Int of int
   | Cond of (boolexp, intexp, intexp)
// end of [intexp]
```

Splitting Recursive Type Declarations (2)

```
datatype boolexp =
   | Bool of bool
   | IntEq of (intexp_t, intexp_t)

assume boolexp_t = boolexp

//
// code for manipulating values of the type [boolexp]
//
```

Splitting Recursive Type Declarations (3)

```
datatype intexp =
   | Int of int
   | Cond of (boolexp_t, intexp, intexp)

assume intexp_t = intexp

//
// code for manipulating values of the type [intexp]
//
```

Related Work

- Barbara Liskov, Abstraction and Specification, MIT Press, 1985.
- John Mitchell and Gordon Plotkin, Abstract types have existential type (POPL'85)
- David MacQueen, Using Dependent Types to Express Modular Structure, (POPL'86)
- Sheldon and Gifford: Static Dependent Types for Modules (Lisp and Functional Programming 1990)
- Harper and Lillibridge: Translucent sums (POPL'94)
- Xavier Leroy: Manifest types (POPL'94)

Conclusion

- A theoretical justification for abstract types is given that is based on conditional type equality
- This justification properly addresses the problem stemmed from directly treating abstract types as existential type
- This justification also obviates the need for complicated mechanisms such as those associated with translucent sums and manifest types