

A Simple and General Theoretical Account for Abstract Types*

Hongwei Xi

Boston University

Abstract. A common approach to hiding implementation details is through the use of abstract types. In this paper, we present a simple theoretical account of abstract types that make use of a recently developed notion of conditional type equality. This is in contrast to most of the existing theoretical accounts of abstract types, which rely on existential types (or similar variants). In addition, we show that this new approach to abstract types opens a promising avenue to the design and implementation of module systems that can effectively support large-scale programming.

1 Introduction

Program organization is a vital issue in the construction of large software. In general, software evolves constantly in order to accommodate emerging needs that are often difficult to foresee, and the ability to effectively localize changes made to the existing programs is of great importance during software development, maintenance and evolution. Most realistic programming languages offer some forms of module system to facilitate the task of partitioning programs into manageable components and then assembling such components into a coherent whole. 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 the other program units and the need for propagating information between program units. The former need helps the construction of a program unit in relative isolation and thus restricts changes in one unit to affect other units while the latter need helps the assembly of program units into a coherent whole.

A common approach to hiding implementation details is through the use abstract types [Lis86, Wir82, CDJ⁺89]. In type theory, existential types [MP85] are often used to give a theoretical account of abstract types. However, there is a rather unpleasant consequence with this account of abstract types. As pointed out long ago (e.g., [Mac86]), hiding type information through existential types often result in too much type information being hidden. In particular, if an existentially quantified package is opened twice, the two abstract type variables thus introduced cannot be assumed equal. As a consequence, an opened existentially quantified package often requires a usage scope so large that most

* Partially supported by NSF grants no. CCR-0229480 and no. CCF-0702665

benefits of abstract types may simply be lost. This issue is certainly of great concern and there have already been many attempts to address it. In most of such attempts, some new forms of types (e.g., dependent types [Mac86], static dependent types [SG90], abstract types via dot notation [CL90], translucent sum types [HL94], manifest types [Ler94]) are introduced to overcome certain limitations of existential types in hiding type information. However, the underlying type theories for these new forms of types are often rather complicated and can become a great deal more complicated if features such as recursive modules [CHP99] and modules as first-class values [Rus00] are to be accommodated.

The primary contribution of the paper lies in a novel theoretic account of abstract types. Instead of relying on existential quantifiers, we make use of recently introduced conditional type equality [Xi04], a seemingly simple notion that we believe is of great potential. Generally speaking, conditional type equality means that type equality is determined under the assumption that certain equations on types hold. For instance, the need for conditional type equality occurs immediately once guarded recursive datatypes are made available [XCC03]. We are to argue that conditional type equality offers an effective means to hiding type information that requires no need for introducing new and unfamiliar forms of types.

We organize the rest of the paper as follows. In Section 2, we form a language λ_2^\supset that supports conditional type equality and then establish the type soundness of λ_2^\supset , presenting a formal account of conditional type equality. We then extend λ_2^\supset to $\lambda_2^\supset+$ to support local binding on abstract type constructors declared at top level. In Section 3, we present some examples to illustrate how certain features of modular programming can be directly supported in $\lambda_2^\supset+$. Lastly, we mention some related work and then conclude. As for a proof of concept, we point out that the programming language ATS [Xi] is currently under active development and its module system, which is largely based on the abstract types presented here, is already functioning.

2 Formal Development

In this section, we first present a language λ_2^\supset , which is largely based upon the standard second-order polymorphically typed λ -calculus, and then extend λ_2^\supset to $\lambda_2^\supset+$ to handle local bindings on abstract type constructors. To simplify the presentation, we only consider quantification over type variables, that is, variables ranging over types, though we also allow quantification over static terms of other sorts (e.g., *bool*, *int*) in ATS. The syntax of λ_2^\supset is given as follows.

$$\begin{array}{c}
\frac{\alpha \in \vec{\alpha}}{\vec{\alpha} \vdash \alpha : \text{type}} \\
\text{TC is } n\text{-ary } \vec{\alpha} \vdash \tau_1 : \text{type} \cdots \vec{\alpha} \vdash \tau_n : \text{type} \\
\hline
\vec{\alpha} \vdash TC(\tau_1, \dots, \tau_n) : \text{type} \\
l_1, \dots, l_n \text{ are distinct } \vec{\alpha} \vdash \tau_1 : \text{type} \cdots \vec{\alpha} \vdash \tau_n : \text{type} \\
\hline
\vec{\alpha} \vdash \{l_1 : \tau_1, \dots, l_n : \tau_n\} : \text{type} \\
\vec{\alpha} \vdash \tau_1 : \text{type} \quad \vec{\alpha} \vdash \tau_2 : \text{type} \\
\hline
\vec{\alpha} \vdash \tau_1 \rightarrow \tau_2 : \text{type} \\
\vec{\alpha}, \alpha \vdash \tau : \text{type} \quad \vec{\alpha}, \alpha \vdash \tau : \text{type} \\
\hline
\vec{\alpha} \vdash \forall \alpha. \tau : \text{type} \quad \vec{\alpha} \vdash \exists \alpha. \tau : \text{type}
\end{array}$$

Fig. 1. The rules for forming types

$$\begin{array}{l}
\text{types } \tau ::= \alpha \mid TC(\tau_1, \dots, \tau_n) \mid \{l_1 : \tau_1, \dots, l_n : \tau_n\} \mid \\
\tau_1 \rightarrow \tau_2 \mid \forall \alpha. \tau \mid \exists \alpha. \tau \\
\text{bindings } B ::= TC(\tau_1, \dots, \tau_n) = \tau \\
\text{exp. } e ::= x \mid f \mid \{l_1 = e_1, \dots, l_n = e_n\} \mid e.l \mid \\
\mathbf{lam} \ x.e \mid \mathbf{fix} \ f.e \mid \mathbf{app}(e_1, e_2) \mid \\
\forall^+(v) \mid \forall^-(e) \mid \exists(e) \mid \mathbf{let} \ \exists(x) = e_1 \ \mathbf{in} \ e_2 \mid \\
\text{values } v ::= x \mid \{l_1 = v_1, \dots, l_n = v_n\} \mid \mathbf{lam} \ x.e \mid \\
\forall^+(v) \mid \exists(v)
\end{array}$$

We use x for a **lam**-variable and f for a **fix**-variable, and xf for either a lam-variable or a fix-variable. A **lam**-variable is a value but a **fix**-variable is not. We use TC for a type constructor of some fixed arity. Also, we use B for a binding of the form $TC(\alpha_1, \dots, \alpha_n) = \tau$ such that the arity of TC is n and $\alpha_1, \dots, \alpha_n \vdash \tau : \text{type}$ is derivable, and say that B is a binding on TC . It is important to notice that for each binding $TC(\alpha_1, \dots, \alpha_n) = \tau$, every free type variable in τ must be α_i for some $1 \leq i \leq n$. We may write TC for $TC()$ if the arity of TC is 0. Also, we write $\vec{\alpha}$ for a sequence of type variables $\alpha_1, \dots, \alpha_n$ and \vec{B} for a sequence of bindings B_1, \dots, B_n , and we use \emptyset for the empty sequence. The rules for forming types are given in Figure 1. In particular, given a type constructor TC of arity n and types τ_1, \dots, τ_n , we can form a type $TC(\tau_1, \dots, \tau_n)$. We use $\{l_1 : \tau_1, \dots, l_n : \tau_n\}$ as a type for labeled records. All other forms of types are standard.

To assign a call-by-value dynamic semantics to expressions in λ_2^\supset , we make use of evaluation contexts, which are defined below:

$$\text{eval. ctx. } E ::= [] \mid E.l \mid \mathbf{app}(E, d) \mid \mathbf{app}(v, E) \mid \\
\forall^-(E) \mid \exists(E) \mid \mathbf{let} \ \exists(x) = E \ \mathbf{in} \ e$$

Definition 1. We define redexes and their reductions as follows.

- $\{l_1 = v_1, \dots, l_n = v_n\}.l_i$ is a redex, and its reduction is v_i , where $1 \leq i \leq n$.

$$\begin{array}{c}
\frac{}{\vec{B} \models \alpha \equiv \alpha} \text{ (tyeq-var)} \\
\frac{\vec{B} \models \tau'_1 \equiv \tau_1 \quad \vec{B} \models \tau_2 \equiv \tau'_2}{\vec{B} \models \tau_1 \rightarrow \tau_2 \equiv \tau'_1 \rightarrow \tau'_2} \text{ (tyeq-}\rightarrow\text{)} \\
\frac{\vec{B} \models \tau \equiv \tau'}{\vec{B} \models \forall \alpha. \tau \equiv \forall \alpha. \tau'} \text{ (tyeq-}\forall\text{)} \\
\frac{\vec{B} \models \tau \equiv \tau'}{\vec{B} \models \exists \alpha. \tau \equiv \exists \alpha. \tau'} \text{ (tyeq-}\exists\text{)} \\
\frac{\vec{B} \models \vec{\tau} \equiv \vec{\tau}'}{\vec{B} \models TC(\vec{\tau}) \equiv TC(\vec{\tau}')} \text{ (tyeq-tc)} \\
\frac{TC(\vec{\alpha}_0) = \tau_0 \text{ is in } \vec{B}}{\vec{B} \models TC(\vec{\tau}) \equiv \tau_0[\vec{\alpha}_0 \mapsto \vec{\tau}]} \text{ (tyeq-unfold)} \\
\frac{TC(\vec{\alpha}_0) = \tau_0 \text{ is in } \vec{B}}{\vec{B} \models \tau_0[\vec{\alpha}_0 \mapsto \vec{\tau}] \equiv TC(\vec{\tau})} \text{ (tyeq-fold)} \\
\frac{\vec{B} \models \tau_1 \equiv \tau_2 \quad \vec{B} \models \tau_2 \equiv \tau_3}{\vec{B} \models \tau_1 \equiv \tau_3} \text{ (tyeq-trans)}
\end{array}$$

Fig. 2. The rules for conditional type equality

- **app**(**lam** $x.e, v$) is a redex, and its reduction is $e[x \mapsto v]$.
- **fix** $f.e$ is redex, and its reduction is $e[f \mapsto \mathbf{fix} f.e]$.
- $\forall^-(\forall^+(v))$ is a redex, and its reduction is v .
- **let** $\exists(x) = \exists(v)$ **in** e is a redex, and its reduction is $e[x \mapsto v]$.

Given two expression e_1 and e_2 such that $e_1 = E[e]$ and $e_2 = E[e']$ for some redex e and its reduction e' , we write $e_1 \hookrightarrow e_2$ and say that e_1 reduces to e_2 in one step. We use \hookrightarrow^* for the reflexive and transitive closure of \hookrightarrow .

The markers $\forall^+(\cdot)$, $\forall^-(\cdot)$ and $\exists(\cdot)$ are mainly introduced to guarantee that the last rule applied in the typing derivation of an expression e be uniquely determined by the structure of e . This in turn makes it significantly easier to establish Theorem 1 (subject reduction) and Theorem 2 (progress). Without these markers, it would be more involved to construct proofs by structural induction on typing derivations.

A typing judgment in $\lambda_2^{\vec{}}$ is of the form $\vec{\alpha}; \Gamma \vdash_{\vec{B}} e : \tau$, which basically means that e can be assigned the type τ under the context $\vec{\alpha}; \Gamma$ if the type equality is decided under \vec{B} through the rules presented in Figure 2. The typing rules for $\lambda_2^{\vec{}}$ are listed in Figure 3, where the obvious side conditions associated with certain rules are omitted. In the following presentation, we may write $\mathcal{D} :: J$ to mean that \mathcal{D} is a derivation for some form of judgment J .

Example 1. Let \vec{B} be $(TC = TC \rightarrow TC)$ for some type constructor TC of arity 0. Then for every pure untyped closed λ -expression e , that is, every closed ex-

$$\begin{array}{c}
\frac{\vec{\alpha}; \Gamma \vdash_{\vec{B}} e : \tau \quad \vec{B} \models \tau \equiv \tau'}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} e : \tau'} \text{ (ty-eq)} \\
\frac{\Gamma(xf) = \tau}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} xf : \tau} \text{ (ty-var)} \\
\frac{\vec{\alpha}; \Gamma \vdash_{\vec{B}} e_1 : \tau_1 \quad \dots \quad \vec{\alpha}; \Gamma \vdash_{\vec{B}} e_n : \tau_n}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} \{l_1 = e_1, \dots, l_n = e_n\} : \{l_1 : \tau_1, \dots, l_n : \tau_n\}} \text{ (ty-rec)} \\
\frac{\vec{\alpha}; \Gamma \vdash_{\vec{B}} e : \{l_1 : \tau_1, \dots, l_n : \tau_n\}}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} e.l_i : \tau_i} \text{ (ty-sel)} \\
\frac{\vec{\alpha}; \Gamma, x : \tau_1 \vdash_{\vec{B}} e : \tau_2}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} \mathbf{lam} x.e : \tau_1 \rightarrow \tau_2} \text{ (ty-lam)} \\
\frac{\vec{\alpha}; \Gamma, f : \tau \vdash_{\vec{B}} e : \tau}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} \mathbf{fix} f.e : \tau} \text{ (ty-fix)} \\
\frac{\vec{\alpha}; \Gamma \vdash_{\vec{B}} e_1 : \tau_1 \rightarrow \tau_2 \quad \vec{\alpha}; \Gamma \vdash_{\vec{B}} e_2 : \tau_1}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} \mathbf{app}(e_1, e_2) : \tau_2} \text{ (ty-app)} \\
\frac{\vec{\alpha}, \alpha; \Gamma \vdash_{\vec{B}} e : \tau}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} \forall^+(e) : \forall \alpha. \tau} \text{ (ty-}\forall^+\text{)} \\
\frac{\vec{\alpha}; \Gamma \vdash_{\vec{B}} e : \forall \alpha. \tau \quad \vec{\alpha} \vdash \tau_0 : \text{type}}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} \forall^-(e) : \tau[\alpha \mapsto \tau_0]} \text{ (ty-}\forall^-\text{)} \\
\frac{\vec{\alpha}; \Gamma \vdash_{\vec{B}} e : \tau[\alpha \mapsto \tau_0] \quad \vec{\alpha} \vdash \tau_0 : \text{type}}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} \exists(e) : \exists \alpha. \tau} \text{ (ty-}\exists^+\text{)} \\
\frac{\vec{\alpha}; \Gamma \vdash_{\vec{B}} e_1 : \exists \alpha. \tau_1 \quad \vec{\alpha}, \alpha; \Gamma, x : \tau_1 \vdash_{\vec{B}} e_2 : \tau_2}{\vec{\alpha}; \Gamma \vdash_{\vec{B}} \mathbf{let} \exists(x) = e_1 \mathbf{in} e_2 : \tau_2} \text{ (ty-}\exists^-\text{)}
\end{array}$$

Fig. 3. The typing rules for $\lambda_2^{\vec{B}}$

pression in $\lambda_2^{\vec{B}}$ that can be constructed in terms of **lam**-variables and **lam** and **app** constructs, the following typing judgment is derivable:

$$\emptyset; \emptyset \vdash_{\vec{B}} e : TC$$

By Theorem 2, which is to be proven shortly, the evaluation of every pure untyped closed λ -expression either terminates or goes on forever; it can never become stuck.

Of course, it is a trivial fact that the evaluation of a pure λ -expression can never become stuck, and Example 1 presents an argument for this fact in $\lambda_2^{\vec{B}}$. Note that \vec{B} in Example 1 is cyclic (according to a definition given later). In general, conditional type equality under a cyclic binding sequence may not be decidable. On the other hand, we are to prove that conditional type equality under an acyclic binding sequence is decidable.

We first show that conditional equality is an equivalence relation.

Proposition 1. *We have the following:*

1. $\vec{B} \models \tau \equiv \tau$ for every type τ , and
2. $\vec{B} \models \tau_1 \equiv \tau_2$ implies $\vec{B} \models \tau_2 \equiv \tau_1$ for every pair of types τ_1, τ_2 .

Therefore, conditional type equality is an equivalence relation by (1) and (2) plus the rule **(tyeq-trans)**.

Proof. By an inspection of the rules in Figure 2.

For every \vec{B} , there is a corresponding term rewriting system $\text{TRS}(\vec{B})$ on types such that for each binding $TC(\vec{\alpha}) = \tau$ in \vec{B} , there is a corresponding rewriting rule $TC(\vec{\alpha}) \Rightarrow \tau$ in $\text{TRS}(\vec{B})$, and we use $\Rightarrow_{\vec{B}}$ for the rewriting relation of $\text{TRS}(\vec{B})$. Obviously, the relation $\vec{B} \models \cdot \equiv \cdot$ is the least equivalence relation containing $\Rightarrow_{\vec{B}}$.

Given a sequence \vec{B} of bindings $TC_1(\vec{\alpha}_1) = \tau_1, \dots, TC_n(\vec{\alpha}_n) = \tau_n$, we say that \vec{B} is linear if TC_1, \dots, TC_n are distinct from each other.

Proposition 2. *If \vec{B} is linear, then $\Rightarrow_{\vec{B}}$ is confluent.*

Proof. If \vec{B} is linear, then there are no critical pairs in $\text{TRS}(\vec{B})$ and thus $\Rightarrow_{\vec{B}}$ is confluent.

Given a type τ , we define $\mathbf{hd}(\tau)$ to be $\{l_1, \dots, l_n\}, \rightarrow, \forall$ and \exists if τ is of the form $\{l_1 : \tau_1, \dots, l_n : \tau_n\}, \tau_1 \rightarrow \tau_2, \forall \alpha. \tau_0$ or $\exists \alpha. \tau$, respectively, and $\mathbf{hd}(\tau)$ is undefined otherwise. Clearly, if $\tau \Rightarrow_{\vec{B}} \tau'$ and $\mathbf{hd}(\tau)$ is defined, then $\mathbf{hd}(\tau') = \mathbf{hd}(\tau)$.

Lemma 1. *Assume \vec{B} is linear and $\vec{B} \models \tau_1 \equiv \tau_2$ holds for some types τ_1 and τ_2 . Then $\mathbf{hd}(\tau_1) = \mathbf{hd}(\tau_2)$ if both $\mathbf{hd}(\tau_1)$ and $\mathbf{hd}(\tau_2)$ are defined.*

Proof. By Proposition 2, we know $\tau_1 \Rightarrow_{\vec{B}} \tau$ and $\tau_2 \Rightarrow_{\vec{B}} \tau$ for some type τ . Assume $\mathbf{hd}(\tau_1) = hd_1$ and $\mathbf{hd}(\tau_2) = hd_2$ for some hd_1, hd_2 . Then $\mathbf{hd}(\tau) = hd_1$ and $\mathbf{hd}(\tau) = hd_2$. Hence, $hd_1 = hd_2$.

The following lemma, which is needed for proving Theorem 2, states that the form of a well-typed closed value is uniquely determined by the type τ of the value if $\mathbf{hd}(\tau)$ is defined.

Lemma 2 (Canonical Forms). *Assume $\mathcal{D} :: \emptyset; \emptyset \vdash_{\vec{B}} v : \tau$ for some linear \vec{B} . Then we have the following:*

1. If $\tau = \{l_1 : \tau_1, \dots, l_n : \tau_n\}$, then v is of the form $\{l_1 = v_1, \dots, l_n = v_n\}$.
2. If $\tau = \tau_1 \rightarrow \tau_2$, then v is of the form **lam** *x.e.*
3. If $\tau = \forall \alpha. \tau_0$, then v is of the form $\forall^+(v_0)$.
4. If $\tau = \exists \alpha. \tau_0$, then v is of the form $\exists(v_0)$.

Proof. Given that the proof is a bit nonstandard, we present some details as follows. In particular, please notice the use of Lemma 1.

We prove (2) by induction on the height of \mathcal{D} . If the last typing rule in \mathcal{D} is **(ty-lam)**, then v is obviously of the form **lam** *x.e.* Otherwise, \mathcal{D} is of the following form,

$$\frac{\mathcal{D}' :: \emptyset; \emptyset \vdash_{\vec{B}} v : \tau' \quad \vec{B} \models \tau' \equiv \tau}{\emptyset; \emptyset \vdash_{\vec{B}} v : \tau} \text{ (ty-eq)}$$

and, by Lemma 1, the following two subcases are the only possibilities:

- $\tau' = \tau'_1 \rightarrow \tau'_2$ such that both $\vec{B} \models \tau'_1 \equiv \tau_1$ and $\vec{B} \models \tau_2 \equiv \tau'_2$ are derivable. Then by induction hypothesis on \mathcal{D}' , we know v is of the form **lam** $x.e$.
- $\tau' = TC(\vec{\tau}')$. Then \mathcal{D}' must be of the form,

$$\frac{\mathcal{D}'' :: \emptyset; \emptyset \vdash_{\vec{B}} v : \tau'' \quad \vec{B} \models \tau'' \equiv \tau'}{\emptyset; \emptyset \vdash_{\vec{B}} v : \tau'} \quad (\mathbf{ty-eq})$$

and therefore, we have the following derivation \mathcal{D}^* as $\vec{B} \models \tau'' \equiv \tau$ holds:

$$\frac{\mathcal{D}'' :: \emptyset; \emptyset \vdash_{\vec{B}} v : \tau'' \quad \vec{B} \models \tau'' \equiv \tau}{\emptyset; \emptyset \vdash_{\vec{B}} v : \tau} \quad (\mathbf{ty-eq})$$

Note that $\mathbf{h}(\mathcal{D}) = 1 + \mathbf{h}(\mathcal{D}^*)$. By induction hypothesis on \mathcal{D}^* , we know that v is of the form **lam** $x.e$.

Hence, (2) holds. (1), (3) and (4) can be proven similarly.

Lemma 3 (Substitution). *We have the following.*

1. Assume that $\vec{\alpha}, \alpha; \Gamma \vdash_{\vec{B}} e : \tau$ and $\vec{\alpha} \vdash \tau_0 : \text{type}$ are derivable. Then $\vec{\alpha}; \Gamma[\alpha \mapsto \tau_0] \vdash_{\vec{B}} e : \tau[\alpha \mapsto \tau_0]$ is derivable.
2. Assume that $\vec{\alpha}; \Gamma, xf : \tau_1 \vdash_{\vec{B}} e_2 : \tau_2$ and $\vec{\alpha}; \Gamma \vdash_{\vec{B}} e_1 : \tau_1$ are derivable. Then $\vec{\alpha}; \Gamma \vdash_{\vec{B}} e_2[xf \mapsto e_1] : \tau_2$ is derivable.

Proof. By structural induction.

We are now ready to establish the soundness of the type system of λ_2^{\supset} by proving the following theorems:

Theorem 1 (Subject Reduction). *Assume $\mathcal{D} :: \emptyset; \emptyset \vdash_{\vec{B}} e : \tau$ in λ_2^{\supset} and $e \hookrightarrow e'$ holds. Then $\emptyset; \emptyset \vdash_{\vec{B}} e' : \tau$ is derivable.*

Proof. Assume that $e = E[e_0]$ and $e' = E[e'_0]$ for some redex e_0 and its reduction e'_0 . The proof proceeds by structural induction on E . In the most interesting case where $E = []$, the proof makes use of Lemma 3.

Theorem 2 (Progress). *Assume $\mathcal{D} :: \emptyset; \emptyset \vdash_{\vec{B}} e : \tau$ in λ_2^{\supset} and \vec{B} is linear. Then e is a value or $e \hookrightarrow e'$ holds for some e' .*

Proof. The theorem follows from structural induction on \mathcal{D} .

We now extend λ_2^{\supset} with a language construct to support type information hiding. We use the name $\lambda_2^{\supset+}$ for this extended language, which contains the following additional syntax for forming expressions:

$$\text{exp. } e ::= \dots \mid \mathbf{assume } B \mathbf{ in } (e : \tau)$$

Note that **assume** ... **in** ... corresponds to the following concrete syntax:

$$\mathbf{local\ assume} \dots \mathbf{in} \dots \mathbf{end}$$

$$\begin{aligned}
|c| &= c \\
|xf| &= xf \\
|\{l_1 = e_1, \dots, l_n = e_n\}| &= \{|e_1|, \dots, |e_n|\} \\
|\mathbf{lam} \ x.e| &= \mathbf{lam} \ x.|e| \\
|\mathbf{fix} \ f.e| &= \mathbf{fix} \ f.|e| \\
|\mathbf{app}(e_1, e_2)| &= \mathbf{app}(|e_1|, |e_2|) \\
|\forall^+(e)| &= \forall^+(|e|) \\
|\forall^-(e)| &= \forall^-(|e|) \\
|\exists(e)| &= \exists(|e|) \\
|\mathbf{let} \ \exists(x) = e_1 \ \mathbf{in} \ e_2| &= \mathbf{let} \ \exists(x) = |e_1| \ \mathbf{in} \ |e_2| \\
|\mathbf{assume} \ B \ \mathbf{in} \ (e : \tau)| &= |e| \\
\mathbf{B}(c) &= \emptyset \\
\mathbf{B}(xf) &= \emptyset \\
\mathbf{B}(\{l_1 = e_1, \dots, l_n = e_n\}) &= \mathbf{B}(e_1), \dots, \mathbf{B}(e_n) \\
\mathbf{B}(\mathbf{lam} \ x.e) &= \mathbf{B}(e) \\
\mathbf{B}(\mathbf{fix} \ f.e) &= \mathbf{B}(e) \\
\mathbf{B}(\mathbf{app}(e_1, e_2)) &= \mathbf{B}(e_1), \mathbf{B}(e_2) \\
\mathbf{B}(\forall^+(e)) &= \mathbf{B}(e) \\
\mathbf{B}(\forall^-(e)) &= \mathbf{B}(e) \\
\mathbf{B}(\exists(e)) &= \mathbf{B}(e) \\
\mathbf{B}(\mathbf{let} \ \exists(x) = e_1 \ \mathbf{in} \ e_2) &= \mathbf{B}(e_1), \mathbf{B}(e_2) \\
\mathbf{B}(\mathbf{assume} \ B \ \mathbf{in} \ (e : \tau)) &= B, \mathbf{B}(e)
\end{aligned}$$

Fig. 4. Two functions on expressions in $\lambda_2^{\exists+}$

We define two functions $|\cdot|$ and $\mathbf{B}(\cdot)$ as in Figure 4. Given an expression e in $\lambda_2^{\exists+}$,

- $|e|$ is the expression in λ_2^{\exists} obtained from erasing in e all the local bindings on abstract type constructors, and
- $\mathbf{B}(e)$ returns a sequence consisting of all the local bindings on abstract type constructors that occur in e , from left to right.

Given a sequence of bindings \vec{B} , then $\mathbf{dom}(\vec{B})$ is a sequence of type constructors \overline{TC} defined as follows:

$$\begin{aligned}
\mathbf{dom}(\emptyset) &= \emptyset \\
\mathbf{dom}(TC(\vec{\alpha} = \tau, \vec{B})) &= TC, \mathbf{dom}(\vec{B})
\end{aligned}$$

We say \overline{TC} is linear if any TC can occur at most once in \overline{TC} .

The typing rules for $\lambda_2^{\exists+}$ are given in Figure 5, where a typing judgment is of the form $\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} e : \tau$ such that $\mathbf{dom}(\mathbf{B}(e)) = \overline{TC}$. Note that the obvious side conditions associated with certain rules are omitted. We use \vec{B} for a sequence of bindings B and \overline{TC} for a sequence of type constructors TC . Please note that the occurrence of \overline{TC} in a typing judgment $\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} e : \tau$ is necessary for supporting separate type-checking as it may not be realistic to assume that (the source code of) e is always available for computing \overline{TC} .

$$\begin{array}{c}
\frac{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} e : \tau \quad \vec{B} \models \tau \equiv \tau'}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} e : \tau'} \text{ (ty-eq)} \\
\frac{\Gamma(xf) = \tau}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\emptyset} xf : \tau} \text{ (ty-var)} \\
\frac{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}_1} e_1 : \tau_1 \quad \dots \quad \vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}_n} e_n : \tau_n \quad \overline{TC} = (\overline{TC}_1, \dots, \overline{TC}_n) \text{ is linear}}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} \{l_1 = e_1, \dots, l_n = e_n\} : \{l_1 : \tau_1, \dots, l_n : \tau_n\}} \text{ (ty-rec)} \\
\frac{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} e : \{l_1 : \tau_1, \dots, l_n : \tau_n\}}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} e.l_i : \tau_i} \text{ (ty-sel)} \\
\frac{\vec{\alpha}; \vec{B}; \Gamma, x : \tau_1 \vdash_{\overline{TC}} e : \tau_2}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} \mathbf{lam} \ x.e : \tau_1 \rightarrow \tau_2} \text{ (ty-lam)} \\
\frac{\vec{\alpha}; \vec{B}; \Gamma, f : \tau \vdash_{\overline{TC}} e : \tau}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} \mathbf{fix} \ f.e : \tau} \text{ (ty-fix)} \\
\frac{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}_1} e_1 : \tau_1 \rightarrow \tau_2 \quad \vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}_2} e_2 : \tau_1 \quad (\overline{TC}_1, \overline{TC}_2) \text{ is linear}}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}_1, \overline{TC}_2} \mathbf{app}(e_1, e_2) : \tau_2} \text{ (ty-app)} \\
\frac{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} e : \tau}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} \forall^+(e) : \forall \alpha. \tau} \text{ (ty-}\forall^+\text{)} \\
\frac{\vec{\alpha}, \alpha; \vec{B}; \Gamma \vdash_{\overline{TC}} e : \forall \alpha. \tau \quad \vec{\alpha} \vdash \tau_0 : \text{type}}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} \forall^-(e) : \tau[\alpha \mapsto \tau_0]} \text{ (ty-}\forall^-\text{)} \\
\frac{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} e : \tau[\alpha \mapsto \tau_0] \quad \vec{\alpha} \vdash \tau_0 : \text{type}}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} \exists(e) : \exists \alpha. \tau} \text{ (ty-}\exists^+\text{)} \\
\frac{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}_1} e_1 : \exists \alpha. \tau_1 \quad \vec{\alpha}, \alpha; \vec{B}; \Gamma, x : \tau_1 \vdash_{\overline{TC}_2} e_2 : \tau_2 \quad (\overline{TC}_1, \overline{TC}_2) \text{ is linear}}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}_1, \overline{TC}_2} \mathbf{let} \ \exists(x) = e_1 \ \mathbf{in} \ e_2 : \tau_2} \text{ (ty-}\exists^-\text{)} \\
\frac{\vec{\alpha}; \vec{B}; TC(\vec{\alpha}_0) = \tau_0; \Gamma \vdash_{\overline{TC}} e : \tau \quad (TC, \overline{TC}) \text{ is linear}}{\vec{\alpha}; \vec{B}; \Gamma \vdash_{TC, \overline{TC}} \mathbf{assume} \ TC(\vec{\alpha}_0) = \tau_0 \ \mathbf{in} \ (e : \tau) : \tau} \text{ (ty-assume)}
\end{array}$$

Fig. 5. The typing rules for $\lambda_2^{\exists}+$

Theorem 3. Assume $\mathcal{D} :: \vec{\alpha}; \vec{B}; \Gamma \vdash_{\overline{TC}} e : \tau$ in $\lambda_2^\supset +$ such that $(\text{dom}(\vec{B}), \overline{TC})$ is linear. Then $\vec{\alpha}; \Gamma \vdash_{(\vec{B}, \mathbf{B}(e))} |e| : \tau$ is derivable in λ_2^\supset .

Proof. By structural induction on \mathcal{D} .

Theorem 3 is the main technical result of the paper, which provides a simple and clean theoretical account of abstract types that requires no use of existential types.¹ We emphasize that the binding sequence \vec{B} in Theorem 3 is only required to be linear (so that Lemma 1 can be established). In particular, because \vec{B} is allowed to be cyclic, Theorem 3 cannot be proven by simply “expanding out” the bindings in \vec{B} .

By Theorem 3, if $\emptyset; \emptyset \vdash_{\overline{TC}} e : \tau$ is derivable in $\lambda_2^\supset +$ for some linear sequence of type constructors \overline{TC} , then $|e|$ can be assigned the type τ in λ_2^\supset under a linear sequence of bindings $\mathbf{B}(e)$. Therefore, by Theorem 2 and Theorem 1, the evaluation of $|e|$ either terminates with a value or goes on forever; it can never become stuck.

We say that \vec{B} is acyclic if \vec{B} is empty or $\vec{B} = (\vec{B}', TC(\alpha) = \tau)$ such that \vec{B}' is acyclic and TC has no occurrences in either τ or \vec{B}' ; otherwise, \vec{B} is cyclic. Given a binding and a type τ , the type $\tau[B]$ is defined as follows:

$$\begin{aligned} \{l_1 : \tau_1, \dots, l_n : \tau_n\}[B] &= \{l_1 : \tau_1[B], \dots, l_n : \tau_n[B]\} \\ (\tau_1 \rightarrow \tau_2)[B] &= \tau_1[B] \rightarrow \tau_2[B] \\ (\forall \alpha. \tau)[B] &= \forall \alpha. \tau[B] \\ (\exists \alpha. \tau)[B] &= \exists \alpha. \tau[B] \\ TC(\vec{\tau})[B] &= \begin{cases} \tau'[\vec{\alpha} \mapsto \vec{\tau}[B]] & \text{if } B \text{ is } TC(\vec{\alpha}) = \tau'; \\ TC(\vec{\tau}[B]) & \text{otherwise.} \end{cases} \end{aligned}$$

Furthermore, given a sequence of bindings \vec{B} and a type τ , the type $\tau[\vec{B}]$ is defined as follows:

$$\tau[\vec{B}] = \begin{cases} \tau & \text{if } \vec{B} = \emptyset; \\ \tau[B][\vec{B}'] & \text{if } \vec{B} = \vec{B}', B. \end{cases}$$

Proposition 3. Assume \vec{B} is an acyclic sequence of bindings. Then $\vec{B} \models \tau_1 \equiv \tau_2$ if and only if $\tau_1[\vec{B}] = \tau_2[\vec{B}]$ holds.

Proof. Straightforward.

Therefore, conditional type equality under an acyclic sequence of bindings can be readily decided. As a design choice, we may simply not allow the use cyclic binding sequences and thus guarantee the decidability of conditional type equality. Whether this choice is too restrictive to support some useful programming styles still needs to be investigated further. We have so far encountered no realistic cases where cyclic sequences of bindings are needed. An argument for this can probably be made as follows. Note that typing the code like the following does not involve cyclic binding sequences:

¹ Note that the existential types in λ_2^\supset are not used to represent abstract types and they can be completely eliminated if one wants to.

```

local assume TC1 = TC2 -> TC2 in ... end
local assume TC2 = TC1 -> TC1 in ... end

```

as the two bindings can never be joined together for deciding type equality. A probably more convincing argument is that we can readily handle the need for splitting mutually defined datatypes, as is to be shown in the next section, with no need for cyclic binding sequences.

Given an expression e in λ_2^\supset , there is in general no principal type for e . For instance, in the following example:

```

abstract PairType: (type, type) -> type

local
  assume PairType (a1, a2) = a2 * a1
in
  fun makePair (x: a1, y: a2) = (y, x)
  ...
end

```

the function *makePair* can be given either the type $\forall\alpha_1.\forall\alpha_2.\alpha_1 * \alpha_2 \rightarrow \alpha_2 * \alpha_1$ or the type $\forall\alpha_1.\forall\alpha_2.\alpha_1 * \alpha_2 \rightarrow PairType(\alpha_1, \alpha_2)$, which are considered equivalent in the scope of the binding $PairType(\alpha_1, \alpha_2) = \alpha_2 * \alpha_1$; however, these two types become unrelated out of the scope. In such a case, it is the responsibility of the programmer to determine through the use of type annotation which type should be assigned to *makePair*.²

3 Examples

In practice, we have frequently encountered the demand for recursive modules, which are unfortunately not supported in the module system of SML [MTHM97]. When forming recursive modules, we often need to split recursively defined datatypes. For instance, we first present a definition for two datatypes *boolexp* and *intexp* in Figure 6 that are intended for representing boolean and integer expressions, respectively. We then show how two abstract types *boolexp_t* and *intexp_t* can be introduced to split the definition into two. In practice, we may declare *boolexp_t* and *intexp_t* in a header file and put the two new definitions in two other files. If, say, the definition of *intexp* needs to be modified later, the modification cannot affect the definition of *boolexp*. While this approach to splitting the definition of mutually defined datatypes may look exceedingly simple, it does not seem simple at all to justify the approach through the use of existential types (or similar variants). On the other hand, a justification based on conditional type equality can be given straightforwardly. Furthermore, it is probably fair to say now that the notion of conditional type equality can also significantly influence the design of module systems.

² In an implementation, one may use a strategy that always expands an abstract type constructor if no type annotation is given. In the case of *makePair*, this means that the type $\forall\alpha_1.\forall\alpha_2.\alpha_1 * \alpha_2 \rightarrow \alpha_2 * \alpha_1$ should be assigned to *makePair*.

```

datatype boolexp =
  | Bool of bool | IntEq of (intexp, intexp)

and intexp =
  | Int of int | Cond of (boolexp, intexp, intexp)

// in the file exp.sats

abstype boolexp_t
abstype intexp_t

// in the file boolexp.dats

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

assume boolexp_t = boolexp

// in the file intexp.dats

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

assume intexp_t = intexp

```

Fig. 6. A splitting of mutually defined datatypes

4 Related Work and Conclusion

There have been a large number of proposals for modular programming, and it is evidently impossible for us to mention even a moderate portion of these proposals here. Instead, we focus on the line of work centered around the module system of Standard ML (SML) [MTHM97], which primarily aims at setting up a theoretical foundation for modular programming based on type theories.

Type abstraction, which can be used to effectively isolate or localize changes made to existing programs, has now become one of the most widely used techniques in specifying and constructing large software systems [GHW85,Lis86]. In [MP88], a theoretical account of abstract types is given through the use of existential types. While this account is largely satisfactory in explaining the features of abstract types, it does not properly address the issue of type equality involving abstract types. Motivated by problems with existential types in modeling abstract types, MacQueen [Mac86] proposed an alternative approach in which abstract types are modeled as a form of dependent sum types, taking into account the issue of type equality involving abstract types. However, in the presence of such dependent sum types, type equality is greatly complicated, and it becomes even more complicated when features such as polymorphism, generativity, higher-order modules and modules as first-class values need to be

accommodated. As a compromise between existential types and dependent sum types, translucent sum types [HL94,Lil97] and manifest types [Ler94] were proposed to better address the issue of hiding type information. There is already a considerable amount of work that studies how these new forms of types can be used to address generativity and applicativity, higher-order modules, recursive modules, modules as first-class values, etc. [Ler95,DCH03,CHP99,Rus01]. There is also work on expressing modular structures in SML and Haskell with only existential types [Rus00,SJ02].

Though the notion of conditional type equality seems surprisingly simple, it had not been recognized in clear terms until recently. We first encountered the notion of conditional type equality in a recent study on guarded recursive datatypes [XCC03].³ This notion has since been generalized in the framework Applied Type System [Xi04].

The major contribution of the paper lies in the recognition and the formalization of a simple and clean theoretic account of abstract types that is based on conditional type equality. In particular, we make no use of existentially quantified types in this account of abstract types, thus completely avoiding the well-known problems associated with the use of existentially quantified types in modeling abstract types. We claim that the presented approach to hiding type information through conditional equality is simple as well as general and it opens a promising avenue for the design and implementation of module systems in support of large-scale programming. As for a proof of concept, we point out that the module system of ATS, which is largely based on $\lambda_2^{\exists,+}$, is already functioning.⁴

Acknowledgments The author acknowledges some discussion with Chiyan Chen, Assaf Kfoury, Likai Liu, Mark Sheldon and Franklyn Turbak on the subject of the paper. The research conducted in this paper has been supported in part by NSF grants no. CCR-0229480 and no. CCF-0702665.

References

- [CDJ⁺89] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalso, and Grep Nelson. The Modula-3 Type System. In *Proceedings of 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 202–212. Austin, TX, January 1989.
- [CHP99] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, pages 56–63, June 1999.
- [CL90] Luca Cardelli and Xavier Leroy. Abstract Types and the Dot Notation. Technical Report 56, DEC SRC, 1990.

³ In Dependent ML (DML) [XP99,Xi98], the type equality may also be classified as conditional type equality. However, the conditions involved are on type indexes rather than on types as in the case of guarded recursive datatypes.

⁴ The current implementation of ATS is named *Anairiats*, which is documented and freely accessible to the public [Xi].

- [DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A Type System for Higher-Order Modules. In *Proceedings of 30th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '03)*, pages 236–249. New Orleans, LA, January 2003.
- [GHW85] J. V. Guttag, J. J. Horning, and Jeannette M. Wing. The Larch Family of Specification Languages. *IEEE Software*, 2(5):24–36, September 1985.
- [HL94] Robert W. Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of 21st Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '94)*, pages 123–137. Portland, Oregon, 1994.
- [Ler94] Xavier Leroy. Manifest Types, Modules, and Separate Compilation. In *Proceeding of 21st Annual ACM Conference on Principles of Programming Languages (POPL '94)*. Portland, OR, January 1994.
- [Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of 22nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '95)*, pages 142–153. San Francisco, CA, January 1995.
- [Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. Ph. D. dissertation, Carnegie Mellon University, May 1997.
- [Lis86] Barbara Liskov. *Abstraction and Specification in Program Development*. MIT Press, 1986.
- [Mac86] David B. MacQueen. Using Dependent Types to Express Modular Structure. In *Proceeding of 13th Annual ACM Symposium on Principles of Programming Languages*, pages 277–286. St. Petersburg Beach, FL, 1986.
- [MP85] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. In *Proceedings of 12th Annual ACM Symposium on Principles of Programming Languages (POPL '85)*, pages 37–51. New Orleans, Louisiana, 1985.
- [MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.
- [MTHM97] Robin Milner, Mads Tofte, Robert W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997. ISBN 0-262-63181-4.
- [Rus00] Claudio V. Russo. First-Class Structures for Standard ML. *Nordic Journal of Computing*, 7(4):348–374, 2000.
- [Rus01] Claudio V. Russo. Recursive Structures for Standard ML. In *Proceedings of International Conference on Functional Programming*, pages 50–61, September 2001.
- [SG90] Mark A. Sheldon and David K. Gifford. Static Dependent Types for First-Class Modules. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 20–29, 1990.
- [SJ02] Mark Shields and Simon Peyton Jones. First-class modules for haskell. In *Proceedings of 9th International Workshop on Foundations of Object-Oriented Languages (FOOL 9)*. Portland, OR, January 2002.
- [Wir82] Niklaus Wirth. *Programming with Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 1982.
- [XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235. ACM press, New Orleans, LA, January 2003.

- [Xi] Hongwei Xi. The ATS Programming Language. Available at:
<http://www.ats-lang.org/>.
- [Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. viii+181 pp. pp. viii+189. Available at:
<http://www.cs.cmu.edu/~hwxi/DML/thesis.ps>.
- [Xi04] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.
- [XP99] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. ACM press, San Antonio, Texas, January 1999.