プログラミング－言語・基礎・実践 12－3
（１９９３．５．２８）

不等式を持つデータ型の性質

金藤栄孝[1]

〒３５０－０３ 埼玉県比企郡鳩山町赤沼２５２０
（株）日立製作所基礎研究所

Abstract

本報告では、Cardelli (1984) のレコード型の体系に不等式の表明を追加して、継承を表明の強さをも考慮するように拡張した型体系を提案し、それが、元の Cardelli の体系の保存的拡大であることを示す。次に、その体系に対して、cpo 上の完備な部分同値関係による意味論を与え、この意味論に対して、本報告の型体系が健全であることを示す。

Some Properties of Data Types with Inequations (Extended Abstract)

Hidetaka Kondoh

Advanced Research Laboratory, Hitachi, Ltd.
Hatoyama, Saitama 350–03, Japan.

Abstract

We start from the Cardelli's work on multiple inheritances and point out that his modeling of data types fails to treat algebraic structures with a concrete example. Then we propose the notion of *algebraic types* by enriching the record type system with inequations and the notion of *algebraic inheritances*, extension of the multiple inheritances à la Cardelli by incorporating the richness of structures, and show our type system is a conservative extension of Cardelli's one by purely syntactical way. Next we give a denotational semantics of our type system on the basis of the complete partial equivalence relation model on a cpo and show that our type system is sound with respect to this semantics.

[1]e-mail: kondoh@harl.hitachi.co.jp

1. Introduction

There are essentially two approaches to formal modeling of abstract data types (*ADTs*): one is from logic using typed λ-calculi; the other is from algebra using first-order equational logic.

An ADT hides two kinds of information; one is the type of the representation of the data structure to be abstracted by that ADT, the *representation type*; the other is a suite of implementations of *associated operations* with that ADT well-typed with respect to the representation type chosen for that ADT, and we call the suite of types of associated operations for an ADT its *implementation type*.

The logical approach uses record types as the basic tool for modeling ADTs. The correspondences between ADTs and record types are summarized as:

implementation type	record type
associated operator symbol	record field label
suite of associated operations	record value
associated operation	value bound to a record field label
inheritance relation	subtype relation on record types

As Reynolds has pointed out in [Reynolds 85], algebras corresponding to record types are only *anarchic* ones ignoring *algebraic structures* of ADTs. In this paper we approach from the logical side and propose a type system incorporating inequations as *assertions* with the record type calculus à la Cardelli [Cardelli 84] corresponding to *non-anarchic* algebra-like structures, hence we call the latter *algebraic types*. Note that due to the limitation of the space, we omit all proofs in this extended abstract.

2. The Cardelli's Approach and Its Problem

The syntax of Cardelli's mini-language, μFun, is given in Fig. 1.

$e \in \mathbf{Exp}$	The set of expressions:
$e ::= x$	(ordinary) variables,
$\mid c$	constants,
$\mid \lambda x{:}\sigma.e$	abstractions,
$\mid e_1 e_2$	applications,
$\mid \{l_1 = e_1, \ldots, l_n = e_n\}$	record expressions ($n \geq 0$),
$\mid e.l$	field selections,
$\mid [l = e]$	tagging expressions,
$\mid$ **case** e **of** l_1 **then** $e_1, \ldots, l_n$ **then** e_n	tag-case expressions ($n \geq 0$),
$\mid \mathbf{fix}(e)$	recursions by fixed-points.
$\sigma, \tau \in \mathbf{Type}$	The set of types:
$\sigma ::= \iota$	basic types,
$\mid \sigma_1 \to \sigma_2$	functional types,
$\mid \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}$	record types ($n \geq 0$),
$\mid [l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n]$	variant types ($n \geq 0$).

Note: We leave details of the following syntactic categories unspecified:

$x \in \mathbf{Var}$	The set of variables;
$c \in \mathbf{Const}$	The set of constant symbols;
$l \in \mathbf{Label}$	The set of record field labels and variant tags;
$\iota \in \mathbf{BaseType}$	The finite set of base types (in Section 5 we assume $\mathbf{BaseType} = \{Bool, Nat\}$).

Figure 1. Syntax of μFun.

The judgment of the subtype relation is

$$\sigma \leq \tau$$

and the relation is defined by:

$$\{\text{BASE}} \quad \iota \leq \iota$$

$$\{\text{TRANS}\} \quad \frac{\sigma_1 \leq \sigma_2 \qquad \sigma_2 \leq \sigma_3}{\sigma_1 \leq \sigma_3}$$

$$\{\text{ARROW}\} \quad \frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\sigma \to \tau \;\leq\; \sigma' \to \tau'}$$

$$\{\text{RECORD}\} \quad \frac{\sigma_1 \leq \tau_1 \;\ldots\; \sigma_n \leq \tau_n}{\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n, \ldots, l_{n+m}{:}\sigma_{n+m}\} \;\leq\; \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}}$$

$$\{\text{VARIANT}\} \quad \frac{\sigma_1 \leq \tau_1 \;\ldots\; \sigma_n \leq \tau_n}{[l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n] \;\leq\; [l_1{:}\tau_1, \ldots, l_n{:}\tau_n, \ldots, l_{n+m}{:}\tau_{n+m}]}$$

Figure 2. The Subtyping Axioms and the Rules of μFun.

$$[\text{VAR}] \quad \Gamma[x:\sigma] \rhd x:\sigma$$

$$[\text{CONST}] \quad \Gamma \rhd c_{ij} : \iota_i$$

$$[\text{WEAK}] \quad \frac{\Gamma \rhd e:\sigma}{\Gamma[x:\sigma'] \rhd e:\sigma} \qquad (x \notin \text{FV}(e))$$

$$[\text{SUBTYPE}] \quad \frac{\Gamma \rhd e:\sigma \qquad \sigma \leq \sigma'}{\Gamma \rhd e:\sigma'}$$

$$[\text{ABS}] \quad \frac{\Gamma[x:\sigma] \rhd e:\sigma'}{\Gamma \rhd (\lambda x{:}\sigma.e) : \sigma \to \sigma'}$$

$$[\text{APPL}] \quad \frac{\Gamma \rhd e:\sigma' \to \sigma \qquad \Gamma \rhd e':\sigma'}{\Gamma \rhd (ee') : \sigma}$$

$$[\text{RECORD}] \quad \frac{\Gamma \rhd e_1:\sigma_1 \;\ldots\; \Gamma \rhd e_n:\sigma_n}{\Gamma \rhd \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}}$$

$$[\text{SELECT}] \quad \frac{\Gamma \rhd e:\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}}{\Gamma \rhd e.l_i : \sigma_i} \qquad (1 \leq i \leq n)$$

$$[\text{VARIANT}] \quad \frac{\Gamma \rhd e:\sigma}{\Gamma \rhd [l = e] : [l{:}\sigma]}$$

$$[\text{CASE}] \quad \frac{\Gamma \rhd e:[l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n] \qquad \Gamma \rhd e_1:\sigma_1 \to \sigma \;\ldots\; \Gamma \rhd e_n:\sigma_n \to \sigma}{\Gamma \rhd (\textbf{case } e \textbf{ of } l_1 \textbf{ then } e_1, \ldots, l_n \textbf{ then } e_n) : \sigma}$$

$$[\text{FIX}] \quad \frac{\Gamma \rhd e:\sigma \to \sigma}{\Gamma \rhd \text{fix}(e) : \sigma}$$

Figure 3. The Typing Axioms and Rules of μFun.

The judgment of typing in μFun is of the form:

$$\Gamma \rhd e:\sigma$$

where Γ is a *context*, i.e. a finite map from variables to types. We introduce notations for contexts:

Notation 1.

(1) $\langle\rangle$ denotes the empty context.

(2) Let Γ be a context, x be a variable, and σ be a type. Then $\Gamma[x:\sigma]$ is the context defined by the following finite map, Γ', such that for any variable y,

$$\Gamma'(y) = \begin{cases} \sigma, & (\text{if } x \equiv y) \\ \Gamma(y). & (\text{otherwise}) \end{cases}$$

(3) The notation, $\mathrm{dom}(\Gamma)$, denotes the set of variables on which the context Γ is defined.

The typing axioms and rules of μFun are given in Fig. 3. where $\mathrm{FV}(e)$ denotes the set of free variables in e.

Now we give a concrete example of the problem in Cardelli's modeling of *multiple inheritances* on ADTs. To display examples compactly, we informally use Standard ML like syntax [MTH 90] for global definitions.

Example. *Stack* (of natural numbers) has as its equipped operations such as: *new* to create a empty stack, *isnew* to check a stack of its emptiness, *push* to add some number to a stack, *top* to see the the top (= lastly pushed) element, *pop* to remove the top element from a stack. Suppose we have *List* as a standard type constructor, and *Nat* and *Bool* as base types in μFun, and we select the list of natural numbers as the representation type for the ADT *Stack*, i.e.

type $StackValRep = List[Nat]$;

Then we can define the implementation type of *Stack* as follows:

$$\begin{aligned} \textbf{type } StackOpImpl = \{&new\colon StackValRep,\\ &isnew\colon StackValRep \to Bool,\\ &push\colon Nat \to StackValRep \to StackValRep,\\ &top\colon StackValRep \to Nat,\\ &pop\colon StackValRep \to StackValRep\}; \end{aligned}$$

Now we can give a suite of implementations of equipped operations of the type *Stack* as a record expression as follows.

$$\begin{aligned} \textbf{val } aStackOpImpl = \{&new = nil,\\ &isnew = \lambda s\colon StackValRep.isnull(s),\\ &push = \lambda i\colon Nat.\lambda s\colon StackValRep.cons(i)(s),\\ &pop = \lambda s\colon StackValRep.tail(s),\\ &top = \lambda s\colon StackValRep.head(s)\}; \end{aligned}$$

This behaves in the *last-in first-out* manner as expected for stacks. E.g. the expression

$$\begin{aligned} &aStackOpImpl.top(\\ &\quad aStackOpImpl.pop(aStackOpImpl.push(2)(aStackOpImpl.push(1)(aStackOpImpl.new)))) \end{aligned}$$

yields 1. On the other hand, with the following suite of implementations

$$\begin{aligned} \textbf{val } anotherStackOpImpl = \{&new = nil,\\ &isnew = \lambda s\colon StackValRep.isnull(s),\\ &push = \lambda i\colon Nat.\lambda s\colon StackValRep.cons(i)(s),\\ &pop = \mathrm{fix}(\lambda p\colon StackValRep \to StackValRep.\lambda s\colon StackValRep.\\ &\qquad \textbf{if } length(s) \le 1 \textbf{ then } nil \textbf{ else } cons(head(s))(p(tail(s)))),\\ &top = \mathrm{fix}(\lambda t\colon StackValRep \to Nat.\lambda s\colon StackValRep.\\ &\qquad \textbf{if } length(s) \le 1 \textbf{ then } head(s) \textbf{ else } t(tail(s)))\}; \end{aligned}$$

where *length* is the usual length function on lists, the value of the expression

$$\begin{aligned} &anotherStackOpImpl.top(anotherStackOpImpl.pop(\\ &\quad anotherStackOpImpl.push(2)(anotherStackOpImpl.push(1)(anotherStackOpImpl.new)))) \end{aligned}$$

is 2, since the *anotherStackOpImpl* acts in the *first-in first-out* fashion. In fact *anotherStackOpImpl* is a suite of implementation adequate for queues rather than for stacks but still has the type *StackOpImpl*.

From this example, we can see that Cardelli's modeling cannot distinguish between behaviors of stacks and of queues, and treats identically stacks and queues having the same type. This limitation is the problem this work attempts to solve.

3. Algebraic Types and Algebraic Inheritances

In the last section, we saw that record types cannot capture all of the aspects of the implementation types of abstract data types. In order to overcome this difficulty we extend the type system of μFun with *inequational assertions* for record types and construct a new language μFinal (μFun with *I*nheritances between *Al*gebraic types). We call such augmented record types as *algebraic types* from an analogy to algebras with equational specifications.

The syntax of μFinal is an extension of that of μFun with following production rules.

$e \in \mathbf{Exp}$	The set of expressions:
$e ::= r$	implementation variables.
$\sigma, \tau \in \mathbf{Type}$	The set of types:
$\sigma ::= \rho r{:}\tau.\{\phi_1, \ldots, \phi_k\}$	algebraic types (τ is a record type, $k \geq 0$).
$\phi, \psi \in \mathbf{Assertion}$	The set of assertions:
$\phi ::= e_1 \leqslant e_2 : \sigma$	atomic assertions,
$\mid$ **forall** $x{:}\sigma.\phi$	quantified assertions.

Note: We leave details of the following syntactic category unspecified:

$r \in \mathbf{IVar}$ The set of implementation variables.

Figure 4. The Characteristic Syntax Rules of μFinal.

Before stating syntactical constraints to μFinal, we need a definition, which is analogous to the notion of *active subexpression* in [Plotkin 77].

Definition 2. Let e, e' be expressions of μFinal. Then e' *strictly occurs in* e iff one of the following conditions holds:

(1) $e \equiv x$ and $e' \equiv x$,
(2) $e \equiv r$ and $e' \equiv r$,
(3) $e \equiv e''.l$ and e' strictly occurs in e'',
(4) $e \equiv e''e'''$ and e' strictly occurs in e'',
(5) $e \equiv$ **case** e'' l_1 **then** $e_1, \ldots, l_n$ **then** e_n and e' strictly occurs in e'',
(6) $e \equiv \mathbf{fix}(e'')$ and e' strictly occurs in e''.

Then the syntactical constraints to μFinal is:

(a) each assertion of an algebraic type must be closed by **forall** quantification except for free occurrences of the implementation variable bound by the algebraic type containing that assertion;
(b) the implementation variable of an algebraic type must strictly occur in the left-hand expression of each assertion of the algebraic type.

Notation 3. We identify each record type with an algebraic type with null assertion, i.e.

$$\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\} \equiv \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\};$$

and we sometimes write algebraic types in more intuitive form, i.e.

$$\rho r.\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n \mid \phi_1, \ldots, \phi_k\} \stackrel{\text{abbrev}}{=} \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_k\}.$$

We often abbreviate symmetrical pair of inequations as an equation, e.g.

$$\textbf{forall } x_1{:}\sigma_1.\cdots.\textbf{forall } x_n{:}\sigma_n.e = e' : \tau \stackrel{\text{abbrev}}{=}$$
$$\textbf{forall } x_1{:}\sigma_1.\cdots.\textbf{forall } x_n{:}\sigma_n.e \leqslant e' : \tau,\ \textbf{forall } x_1{:}\sigma_1.\cdots.\textbf{forall } x_n{:}\sigma_n.e' \leqslant e : \tau.$$

Moreover we will omit implementation variables in assertions and write l for $r.l$ when there is no danger of confusion.

For the type system of μFinal, we introduce a first-order theory of μFinal.

Definition 4.

(1) μFINAL is the first-order theory with axioms and rules which will be described in this section and with the following three forms of judgments as sentences:

- $\sigma \leq \tau$ for subtyping,
- $\Gamma, \Delta \rhd e{:}\sigma$ for typing,
- $\Gamma, \Delta \rhd \phi$ for assertions,

where Δ is a context for implementation variables. The deducibility in μFINAL is shown by $\vdash_{\mu\text{FINAL}}$.

(2) For the type system of μFun, μFUN is defined in the same way, and $\vdash_{\mu\text{FUN}}$ denotes its deducibility.

Note: We usually omit the subscripts and simply write $\vdash$ when there is no danger of confusions.

First we define the subtype relation on μFinal. The {RECORD} rule of μFun is generalized to handle assertions.

$$\text{\{ALGEBRA\}}\quad \frac{\bigwedge_{i=1}^{k}(\langle\rangle, \langle r:\sigma\rangle \rhd \phi_i[r_1 := r]) \vdash_{\mu\text{FINAL}} \bigwedge_{j=1}^{l}(\langle\rangle, \langle r:\tau\rangle \rhd \psi_j[r_2 := r]) \qquad \sigma_1 \leq \tau_1 \ \ldots\ \sigma_n \leq \tau_n}{\rho r_1{:}\{l_1{:}\sigma_1, \ldots, l_{n+m}{:}\sigma_{n+m}\}.\{\phi_1, \ldots, \phi_k\} \ \leq\ \rho r_2{:}\{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}.\{\psi_1, \ldots, \psi_l\}} \quad (m \geq 0)$$

where

$\bigwedge_{i=1}^{k}(\langle\rangle, \langle r:\sigma\rangle \rhd \phi_i[r_1 := r]) \vdash_{\mu\text{FINAL}} \bigwedge_{j=1}^{l}(\langle\rangle, \langle r:\tau\rangle \rhd \psi_j[r_2 := r])$ is a short-hand notation meaning that for each $1 \leq j \leq l$,

$\langle\rangle, \langle r:\sigma\rangle \rhd \phi_1[r_1 := r], \ldots, \langle\rangle, \langle r:\sigma\rangle \rhd \phi_k[r_1 := r] \vdash_{\mu\text{FINAL}} \langle\rangle, \langle r:\tau\rangle \rhd \psi_j[r_2 := r]$;

r is a fresh implementation variable;

$\sigma \equiv \{l_1{:}\sigma_1, \ldots, l_{n+m}{:}\sigma_{n+m}\}$;

$\tau \equiv \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}$.

Figure 5. The Characteristic Subtyping Rule of μFinal.

Intuitively speaking, this {ALGEBRA} rule states that if an algebraic type is a subtype of another one in the sense of record types (i.e. $\sigma \leq \tau$) and the set of assertions of the former, $\{\phi_1, \ldots, \phi_k\}$, is stronger than that of the latter, $\{\psi_1, \ldots, \psi_l\}$, then the former type is a subtype of the latter one as algebraic types.

For typing in μFinal, we replace each judgment of the form $\Gamma \rhd e : \sigma$ in the typing axioms and rules of μFun by one of the form $\Gamma, \Delta \rhd e : \sigma$ augmented with a context for implementation variables. Furthermore, we have to add an axiom and two rules:

$$\text{[IVAR]}\quad \Gamma, \Delta[r:\tau] \rhd r : \tau$$

$$\text{[IWEAK]}\quad \frac{\Gamma, \Delta \rhd e : \sigma}{\Gamma, \Delta[r:\tau] \rhd e : \sigma} \qquad (r \notin \mathrm{FV}(e))$$

$$\text{[EXTEND]}\quad \frac{\Gamma, \Delta \rhd e : \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_k\} \qquad \Gamma, \Delta \rhd \phi_{k+1}[r := e]}{\Gamma, \Delta \rhd e : \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_{k+1}\}}$$

Figure 6. The Characteristic Typing Axiom and the Rules of μFinal.

The [EXTEND] rule means intuitively that if the expression e satisfies the assertion ϕ_{k+1} then we can add this assertion to the algebraic type of e.

Finally we must give the rules for inferring assertions. These rules are shown in Fig. 7. Note that each of the rules, $\langle\beta_{\text{func}}\rangle$, $\langle\beta_{\text{record}}\rangle$, $\langle\beta_{\text{variant}}\rangle$ and $\langle\beta_{\text{fix}}\rangle$, whose conclusion is an equational form actually denotes a pair of rules by the notational convention as stated before, and the meaning of each rule is apparent except for $\langle$ASSERT$\rangle$. This rule states that if an expression, e, has an algebraic type with assertions, $\phi_1, \ldots, \phi_k$, in which the representation variable, r, denotes e, then we can use an instance of each assertion, ϕ_i, by substituting that expression, e, for r.

$$\langle \text{VAR} \rangle \quad \Gamma[x:\sigma], \Delta \triangleright x \leqslant x : \sigma$$

$$\langle \text{IVAR} \rangle \quad \Gamma, \Delta[r:\tau] \triangleright r \leqslant r : \tau$$

$$\langle \text{CONST} \rangle \quad \Gamma, \Delta \triangleright c_{ij} \leqslant c_{ij} : \iota_i$$

$$\langle \text{TRANS} \rangle \frac{\Gamma, \Delta \triangleright e_1 \leqslant e_2 : \sigma \qquad \Gamma, \Delta \triangleright e_2 \leqslant e_3 : \sigma}{\Gamma, \Delta \triangleright e_1 \leqslant e_3 : \sigma}$$

$$\langle \text{WEAK} \rangle \frac{\Gamma, \Delta \triangleright e_1 \leqslant e_2 : \sigma}{\Gamma[x:\sigma'], \Delta \triangleright e_1 \leqslant e_2 : \sigma} \qquad (x \notin \text{FV}(e_1) \cup \text{FV}(e_2))$$

$$\langle \text{IWEAK} \rangle \frac{\Gamma, \Delta \triangleright e_1 \leqslant e_2 : \sigma}{\Gamma, \Delta[r:\tau] \triangleright e_1 \leqslant e_2 : \sigma} \qquad (r \notin \text{FV}(e_1) \cup \text{FV}(e_2))$$

$$\langle \text{SUBTYPE} \rangle \frac{\Gamma, \Delta \triangleright e_1 \leqslant e_2 : \sigma \qquad \sigma \leq \sigma'}{\Gamma, \Delta \triangleright e_1 \leqslant e_2 : \sigma'}$$

$$\langle \textbf{forall-E} \rangle \frac{\Gamma, \Delta \triangleright \textbf{forall}\ x{:}\sigma.\phi \qquad \Gamma, \Delta \triangleright e : \sigma}{\Gamma, \Delta \triangleright \phi[x := e]}$$

$$\langle \textbf{forall-I} \rangle \frac{\Gamma[x:\sigma], \Delta \triangleright \phi}{\Gamma, \Delta \triangleright \textbf{forall}\ x{:}\sigma.\phi} \qquad (x \notin \text{dom}(\Gamma))$$

$$\langle \beta_{\text{func}} \rangle \frac{\Gamma[x:\sigma], \Delta \triangleright e : \sigma' \qquad \Gamma, \Delta \triangleright e' : \sigma}{\Gamma, \Delta \triangleright (\lambda x{:}\sigma.e)e' = e[x := e'] : \sigma'}$$

$$\langle \text{ABS} \rangle \frac{\Gamma[x:\sigma], \Delta \triangleright e \leqslant e' : \sigma'}{\Gamma, \Delta \triangleright (\lambda x{:}\sigma.e) \leqslant (\lambda x{:}\sigma.e') : \sigma \to \sigma'}$$

$$\langle \text{APPL} \rangle \frac{\Gamma, \Delta \triangleright e_1 \leqslant e_1' : \sigma' \to \sigma \qquad \Gamma, \Delta \triangleright e_2 \leqslant e_2' : \sigma'}{\Gamma, \Delta \triangleright (e_1 e_2) \leqslant (e_1' e_2') : \sigma}$$

$$\langle \beta_{\text{record}} \rangle \frac{\Gamma, \Delta \triangleright e_1 : \sigma_1 \ \ldots\ \Gamma, \Delta \triangleright e_n : \sigma_n}{\Gamma, \Delta \triangleright \{l_1 = e_1, \ldots, l_n = e_n\}.l_i = e_i : \sigma_i} \qquad (1 \leq i \leq n)$$

$$\langle \text{RECORD} \rangle \frac{\Gamma, \Delta \triangleright e_1 \leqslant e_1' : \sigma_1 \ \ldots\ \Gamma, \Delta \triangleright e_n \leqslant e_n' : \sigma_n}{\Gamma, \Delta \triangleright \{l_1 = e_1, \ldots, l_n = e_n\} \leqslant \{l_1 = e_1', \ldots, l_n = e_n'\} : \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}}$$

$$\langle \text{SELECT} \rangle \frac{\Gamma, \Delta \triangleright e \leqslant e' : \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}}{\Gamma, \Delta \triangleright e.l_i = e'.l_i : \sigma_i} \qquad (1 \leq i \leq n)$$

$$\langle \text{ASSERT} \rangle \frac{\Gamma, \Delta \triangleright e : \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_k\}}{\Gamma, \Delta \triangleright \phi_i[r := e]} \qquad (1 \leq i \leq k)$$

$$\langle \beta_{\text{variant}} \rangle \frac{\Gamma, \Delta \triangleright e : \sigma_i \qquad \Gamma, \Delta \triangleright e_1 : \sigma_1 \to \sigma' \ \ldots\ \Gamma, \Delta \triangleright e_n : \sigma_n \to \sigma'}{\Gamma, \Delta \triangleright (\textbf{case}\ [l_i = e]\ \textbf{of}\ l_1\ \textbf{then}\ e_1, \ldots, l_n\ \textbf{then}\ e_n) = (e_i e) : \sigma'} \qquad (1 \leq i \leq n)$$

$$\langle \text{VARIANT} \rangle \frac{\Gamma, \Delta \triangleright e \leqslant e' : \sigma}{\Gamma, \Delta \triangleright [l = e] \leqslant [l = e'] : [l{:}\sigma]}$$

$$\langle \text{CASE} \rangle \frac{\Gamma, \Delta \triangleright e \leqslant e' : [l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n] \qquad \Gamma, \Delta \triangleright e_1 \leqslant e_1' : \sigma_1 \to \sigma' \ \ldots\ \Gamma, \Delta \triangleright e_n \leqslant e_n' : \sigma_n \to \sigma'}{\Gamma, \Delta \triangleright (\textbf{case}\ e\ \textbf{of}\ l_1\ \textbf{then}\ e_1, \ldots, l_n\ \textbf{then}\ e_n) \leqslant (\textbf{case}\ e'\ \textbf{of}\ l_1\ \textbf{then}\ e_1', \ldots, l_n\ \textbf{then}\ e_n') : \sigma'}$$

$$\langle \beta_{\text{fix}} \rangle \frac{\Gamma, \Delta \triangleright e : \sigma \to \sigma}{\Gamma, \Delta \triangleright \text{fix}(e) = e(\text{fix}(e)) : \sigma}$$

$$\langle \text{FIX} \rangle \frac{\Gamma, \Delta \triangleright e \leqslant e' : \sigma \to \sigma}{\Gamma, \Delta \triangleright \text{fix}(e) \leqslant \text{fix}(e') : \sigma}$$

Figure 7. The Axioms and the Rules for Assertions of μFinal.

We end this section by giving an example of algebraic inheritances. We can define the implementation type of in the last section as an algebraic type.

$$
\begin{array}{ll}
\textbf{type } \mathit{StackOpImpl} = \rho a\mathit{StackOpImpl}.\{\mathit{new}\colon \mathit{StackValRep}, & \\
\qquad \mathit{isnew}\colon \mathit{StackValRep} \to \mathit{Bool}, & \\
\qquad \mathit{push}\colon \mathit{Nat} \to \mathit{StackValRep} \to \mathit{StackValRep}, & \\
\qquad \mathit{top}\colon \mathit{StackValRep} \to \mathit{Nat}, & \\
\qquad \mathit{pop}\colon \mathit{StackValRep} \to \mathit{StackValRep} & \\
\text{(*push-pop*)} \quad \mid \textbf{forall } i\colon \mathit{Nat}.\textbf{forall } s\colon \mathit{StackValRep}. & \\
\qquad\qquad \mathit{pop}(\mathit{push}(i)(s)) \leqslant s : \mathit{StackValRep}, & \\
\qquad \textbf{forall } i\colon \mathit{Nat}.\textbf{forall } s\colon \mathit{StackValRep}.\mathit{top}(\mathit{push}(i)(s)) \leqslant i : \mathit{Nat}, & \\
\qquad \textbf{forall } i\colon \mathit{Nat}.\textbf{forall } s\colon \mathit{StackValRep}.\mathit{isnew}(\mathit{push}(i)(s)) \leqslant \mathit{false} : \mathit{Bool}, & \\
\qquad \mathit{isnew}(\mathit{new}) \leqslant \mathit{true} : \mathit{Bool}\}; &
\end{array}
$$

Now consider another type *StackoidOpImpl*:

$$
\begin{array}{ll}
\textbf{type } \mathit{StackoidOpImpl} = \rho a\mathit{StackoidOpImpl}.\{\mathit{new}\colon \mathit{StackValRep}, & \\
\qquad \mathit{isnew}\colon \mathit{StackValRep} \to \mathit{Bool}, & \\
\qquad \mathit{push}\colon \mathit{Nat} \to \mathit{StackValRep} \to \mathit{StackValRep}, & \\
\qquad \mathit{top}\colon \mathit{StackValRep} \to \mathit{Nat}, & \\
\qquad \mathit{pop}\colon \mathit{StackValRep} \to \mathit{StackValRep} & \\
\text{(*push}^2\text{-pop}^2\text{*)} \quad \mid \textbf{forall } i\colon \mathit{Nat}.\textbf{forall } j\colon \mathit{Nat}.\textbf{forall } s\colon \mathit{StackValRep}. & \\
\qquad\qquad \mathit{pop}(\mathit{pop}(\mathit{push}(i)(\mathit{push}(j)(s)))) \leqslant s : \mathit{StackValRep}, & \\
\qquad \textbf{forall } i\colon \mathit{Nat}.\textbf{forall } s\colon \mathit{StackValRep}.\mathit{top}(\mathit{push}(i)(s)) \leqslant i : \mathit{Nat}, & \\
\qquad \textbf{forall } i\colon \mathit{Nat}.\textbf{forall } s\colon \mathit{StackValRep}. & \\
\qquad\qquad \mathit{isnew}(\mathit{push}(i)(s)) \leqslant \mathit{false} : \mathit{Bool}, & \\
\qquad \mathit{isnew}(\mathit{new}) \leqslant \mathit{true} : \mathit{Bool}\}; &
\end{array}
$$

Clearly the assertion (*push-pop*) in *StackOpImpl* is stronger than (*push^2-pop^2*) in *StackoidOpImpl*, hence any stack can be used as a stackoid. Therefore *StackOpImpl* inherits the structure of *StackoidOpImpl*, and this fact is expressed in μFinal as the subtype relationship $\mathit{StackOpImpl} \leq \mathit{StackoidOpImpl}$.

Now we show the definition of the type *QueueOpImpl* with the same signature of *StackOpImpl* for comparison:

$$
\begin{array}{ll}
\textbf{type } \mathit{QueueOpImpl} = \rho a\mathit{QueueOpImpl}.\{\mathit{new}\colon \mathit{StackValRep}, & \\
\qquad \mathit{isnew}\colon \mathit{StackValRep} \to \mathit{Bool}, & \\
\qquad \mathit{push}\colon \mathit{Nat} \to \mathit{StackValRep} \to \mathit{StackValRep}, & \\
\qquad \mathit{top}\colon \mathit{StackValRep} \to \mathit{Nat}, & \\
\qquad \mathit{pop}\colon \mathit{StackValRep} \to \mathit{StackValRep} & \\
\qquad \mid \textbf{forall } i\colon \mathit{Nat}.\textbf{forall } s\colon \mathit{StackValRep}. & \\
\qquad\qquad \mathit{pop}(\mathit{push}(i)(s)) \leqslant \textbf{if } \mathit{isnew}(s) \textbf{ then } s & \\
\qquad\qquad\qquad\qquad \textbf{else } \mathit{push}(i)(\mathit{pop}(s)) : \mathit{StackValRep}, & \\
\qquad \textbf{forall } i\colon \mathit{Nat}.\textbf{forall } s\colon \mathit{StackValRep}. & \\
\qquad\qquad \mathit{top}(\mathit{push}(i)(s)) \leqslant \textbf{if } \mathit{isnew}(s) \textbf{ then } i \textbf{ else } \mathit{top}(s) : \mathit{Nat}, & \\
\qquad \textbf{forall } i\colon \mathit{Nat}.\textbf{forall } s\colon \mathit{StackValRep}.\mathit{isnew}(\mathit{push}(i)(s)) \leqslant \mathit{false} : \mathit{Bool}, & \\
\qquad \mathit{isnew}(\mathit{new}) \leqslant \mathit{true} : \mathit{Bool}\}; &
\end{array}
$$

Then clearly $\mathit{StackOpImpl} \not\leq \mathit{QueueOpImpl}$ and $\mathit{QueueOpImpl} \not\leq \mathit{StackOpImpl}$. Moreover, for *aStackOpImpl* and *anotherStackOpImpl* in Section 2, we can show $\vdash_{\mu\text{FINAL}} \langle\rangle, \langle\rangle \triangleright \mathit{aStackOpImpl} : \mathit{StackOpImpl}$ and $\vdash_{\mu\text{FINAL}} \langle\rangle, \langle\rangle \triangleright \mathit{anotherStackOpImpl} : \mathit{QueueOpImpl}$ as we have pointed out in Section 2 (here we assume assertions on list operations are given).

4. Proof Theoretical Investigations of Algebraic Types

In this section we investigate the proof theoretical properties of the type system μFINAL. Especially we show the system is a conservative extension of μFUN.

First, we define classes of types, expressions, contexts and sentences of μFinal having correspondences in μFun.

Definition 5.

(1) A type, σ, of μFinal is said to be *assertion-free* iff one of the following condition holds:
   (a) $\sigma \equiv \iota$;
   (b) $\sigma \equiv \sigma_1 \to \sigma_2$, where each σ_i $(i = 1, 2)$ is assertion-free;
   (c) $\sigma \equiv \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}$, where each σ_i $(1 \leq i \leq n)$ is assertion-free;
   (d) $\sigma \equiv [l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n]$, where each σ_i $(1 \leq i \leq n)$ is assertion-free.

(2) An expression, e, of μFinal is said to be *assertion-free* iff both of the following conditions hold:
   (a) e does not contain any implementation variable,
   (b) each λ-abstraction occurring in e binds a variable with an assertion-free type.

(3) A context, Γ, is *assertion-free* iff Γ assigns assertion-free types to each variable.

(4) A sentence, Σ, of μFINAL is *assertion-free* iff Σ has either one of the following forms:
   (a) $\Sigma \equiv \Gamma, \langle\rangle \rhd e : \sigma$, where Γ, e and σ are assertion-free, respectively;
   (b) $\Sigma \equiv \sigma \leq \tau$, where σ and τ are assertion-free.

For each assertion-free sentence of μFinal, we define the correspondence in μFun.

Definition 6. Let Σ be an assertion-free sentence of μFINAL. Then define its *corresponding sentence in* μFUN (notation: $\mu\mathrm{FUN}(\Sigma)$) as follows:

(1) when $\Sigma \equiv \Gamma, \langle\rangle \rhd e : \sigma$, $\mu\mathrm{FUN}(\Sigma) \equiv \Gamma \rhd e : \sigma$;
(2) when $\Sigma \equiv \sigma \leq \tau$, $\mu\mathrm{FUN}(\Sigma) \equiv \Sigma$.

Next we define a function which removes all assertions from types.

Definition 7. The function aet : **Type** $\to$ **Type** is defined such that

(1) $\mathrm{aet}(\iota) = \iota$,
(2) $\mathrm{aet}(\sigma \to \tau) = \mathrm{aet}(\sigma) \to \mathrm{aet}(\tau)$,
(3) $\mathrm{aet}(\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}) = \{l_1{:}\mathrm{aet}(\sigma_1), \ldots, l_n{:}\mathrm{aet}(\sigma_n)\}$,
(4) $\mathrm{aet}([l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n]) = [l_1{:}\mathrm{aet}(\sigma_1), \ldots, l_n{:}\mathrm{aet}(\sigma_n)]$,
(5) $\mathrm{aet}(\rho\tau{:}\tau.\{\phi_1, \ldots, \phi_k\}) = \mathrm{aet}(\tau)$.

This aet is extended on **Exp**, **Assertion**, contexts and sentences of μFINAL in the obvious way.

Finally we can show the desired result.

Theorem 8. *If an assertion-free sentence Σ is provable in μFINAL, then there is a proof of Σ with only assertion-free sentences.* ∎

As stated before, any assertion-free sentence Σ of μFINAL corresponds to some sentence of μFUN.

Corollary 9 Conservative Extension Theorem. *The theory μFINAL is a conservative extension of μFUN. That is, for any assertion-free sentence Σ of μFINAL,*

$$\vdash_{\mu\mathrm{FINAL}} \Sigma \quad \Longleftrightarrow \quad \vdash_{\mu\mathrm{FUN}} \mu\mathrm{FUN}(\Sigma).$$ ∎

The type system of μFinal is clearly undecidable since it has power to specify a kind of partial correctness of functional programs. But the system restores decidability by forgetting all assertions, hence our system μFINAL can be viewed as a type system for specification/verification while μFUN is its decidable subsystem for compile-time type-checking. Then the following theorem states that "*all correct programs pass compilers.*"

Theorem 10. *For any Γ, Δ, σ, and any e without implementation variables,*

$$\vdash_{\mu\mathrm{FINAL}} \Gamma, \Delta \rhd e : \sigma \quad \Longrightarrow \quad \vdash_{\mu\mathrm{FUN}} \mathrm{aet}(\Gamma) \rhd \mathrm{aet}(e) : \mathrm{aet}(\sigma)..$$ ∎

5. Semantics of Algebraic Types and Algebraic Inheritances

We first give a semantics of expressions using the erasure interpretation. The semantic domain $\mathbf{D}$ for the interpretation is the complete partially ordered set (*cpo*) satisfying the following domain equation (we can find such $\mathbf{D}$ in the universal domain $\mathbf{T}_\perp{}^\omega$ by the well known techniques [Plotkin 78] after appropriate encoding of truth values, natural numbers and labels/tags, and we usually omit the isomorphisms between $\mathbf{D}$ and the right-hand sum cpo). For details on cpos we follow [Plotkin 83] and [Barendregt 81].

$$v \in \mathbf{D} \cong \mathbf{A}_0 \oplus \mathbf{A}_1 \oplus \mathbf{F} \oplus \mathbf{R} \oplus \mathbf{U} \oplus \mathbf{W}$$

where

- $\mathbf{A}_0 = \mathbf{T}_\perp$ and $\mathbf{A}_1 = \mathbf{N}_\perp$,
- $f \in \mathbf{F} = [\mathbf{D} \to \mathbf{D}]$ is for function values;
- $q \in \mathbf{R} = [\mathbf{Label}_\perp \to_\perp \mathbf{D}]$ is for record values;
- $u \in \mathbf{U} = [\mathbf{Label}_\perp \times \mathbf{D}]$ is for variant (tagged union) values;
- $\mathbf{W} \stackrel{\text{def}}{=} \{?\}_\perp$ and $wrong \stackrel{\text{def}}{=} in_{\mathbf{W}}(?)$ representing run-time type errors;

We also need a few auxiliary domains for environments:

$\varepsilon \in \mathbf{Env} = \mathbf{EEnv} \times \mathbf{IEnv}$	the domain of environments;
$\zeta \in \mathbf{EEnv} = \mathbf{Var}_\perp \to_\perp \mathbf{D}$	the domain of valuations for ordinary variables;
$\xi \in \mathbf{IEnv} = \mathbf{IVar}_\perp \to_\perp \mathbf{D}$	the domain of valuations for implementation variables.

The semantic equations for expressions are shown in Fig. 8 (here we assume a semantic function $\mathcal{K}_i$ for each base type ι_i for the interpretations of its constants).

$$\mathcal{E} : \mathbf{Exp} \to (\mathbf{Env} \to \mathbf{D})$$

$$\begin{aligned}
\mathcal{E}[\![x]\!]\varepsilon &= \mathit{let}\ \langle \zeta, \xi \rangle = \varepsilon\ \mathit{in}\ \zeta[\![x]\!]\ \mathit{end};\\
\mathcal{E}[\![r]\!]\varepsilon &= \mathit{let}\ \langle \zeta, \xi \rangle = \varepsilon\ \mathit{in}\ \xi[\![r]\!]\ \mathit{end};\\
\mathcal{E}[\![c_{ij}]\!]\varepsilon &= in_{\mathbf{A}_i}(\mathcal{K}_i[\![c_{ij}]\!]);\\
\mathcal{E}[\![\lambda x{:}\sigma.e]\!]\varepsilon &= \mathit{let}\ \langle \zeta, \xi \rangle = \varepsilon\ \mathit{in}\ in_{\mathbf{F}}(\lambda v \in \mathbf{D}.\mathcal{E}[\![e]\!]\langle \zeta[x \mapsto v], \xi \rangle)\ \mathit{end};\\
\mathcal{E}[\![ee']\!]\varepsilon &= \mathit{if}\ is_{\mathbf{F}}(\mathcal{E}[\![e]\!]\varepsilon)\ \mathit{then}\ out_{\mathbf{F}}(\mathcal{E}[\![e]\!]\varepsilon)(\mathcal{E}[\![e']\!]\varepsilon)\ \mathit{else}\ \mathit{wrong};\\
\mathcal{E}[\![\{l_1 = e_1, \ldots, l_n = e_n\}]\!]\varepsilon &= in_{\mathbf{R}}(\lambda l \in \mathbf{Label}_\perp.\mathit{if}\ l = l_1\ \mathit{then}\ \mathcal{E}[\![e_1]\!]\varepsilon\\
&\qquad \mathit{elseif}\ \ldots\\
&\qquad \mathit{elseif}\ l = l_n\ \mathit{then}\ \mathcal{E}[\![e_n]\!]\varepsilon\\
&\qquad \mathit{else}\ \mathit{wrong});\\
\mathcal{E}[\![e.l]\!]\varepsilon &= \mathit{if}\ is_{\mathbf{R}}(\mathcal{E}[\![e]\!]\varepsilon)\ \mathit{then}\ out_{\mathbf{R}}(\mathcal{E}[\![e]\!]\varepsilon)(l)\ \mathit{else}\ \mathit{wrong};\\
\mathcal{E}[\![[l = e]]\!]\varepsilon &= in_{\mathbf{U}}\langle l, \mathcal{E}[\![e]\!]\varepsilon \rangle;\\
\mathcal{E}[\![\mathbf{case}\ e\ \mathbf{of}\ l_1\ \mathbf{then}\ e_1, \ldots, l_n\ \mathbf{then}\ e_n]\!]\varepsilon &= \mathit{if}\ is_{\mathbf{U}}(\mathcal{E}[\![e]\!]\varepsilon)\ \mathit{then}\\
&\qquad \mathit{let}\ \langle l, v \rangle = out_{\mathbf{U}}(\mathcal{E}[\![e]\!]\varepsilon)\ \mathit{in}\\
&\qquad\quad \mathit{if}\ l = l_1\ \mathit{then}\\
&\qquad\qquad \mathit{if}\ is_{\mathbf{F}}(\mathcal{E}[\![e_1]\!]\varepsilon)\ \mathit{then}\ out_{\mathbf{F}}(\mathcal{E}[\![e_1]\!]\varepsilon)(v)\ \mathit{else}\ \mathit{wrong}\\
&\qquad\quad \mathit{elseif}\ \ldots\\
&\qquad\quad \mathit{elseif}\ l = l_n\ \mathit{then}\\
&\qquad\qquad \mathit{if}\ is_{\mathbf{F}}(\mathcal{E}[\![e_n]\!]\varepsilon)\ \mathit{then}\ out_{\mathbf{F}}(\mathcal{E}[\![e_n]\!]\varepsilon)(v)\ \mathit{else}\ \mathit{wrong}\\
&\qquad\quad \mathit{else}\ \mathit{wrong}\\
&\qquad \mathit{end}\\
&\qquad \mathit{else}\ \mathit{wrong};\\
\mathcal{E}[\![\mathbf{fix}(e)]\!]\varepsilon &= \mathit{if}\ is_{\mathbf{F}}(\mathcal{E}[\![e]\!]\varepsilon)\ \mathit{then}\ \mathit{let}\ f = out_{\mathbf{F}}(\mathcal{E}[\![e]\!]\varepsilon)\ \mathit{in}\ \bigsqcup_n f^n(\perp_{\mathbf{D}})\ \mathit{end}\\
&\qquad \mathit{else}\ \mathit{wrong}.
\end{aligned}$$

Figure 8. The Semantic Equations for Expressions of μFinal.

Next we give a semantics for types based on a kind of partial equivalence models.

Definition 11. Let X be a set.

(1) A *partial equivalence relation* (*per* for short) on X is a symmetric and transitive binary relation on X.

(2) Let P be a per on X. Then define the *domain* of P (notation: $|P|$) by:

$$|P| \stackrel{\text{def}}{=} \{v \in X \mid \langle v, v\rangle \in P\}.$$

(3) Let P and Q be pers on X. Then define the function space per, $P \to Q$, by:

$$\langle f, g\rangle \in P \to Q \iff \forall v, v' \in X.[\langle v, v'\rangle \in P \Rightarrow \langle f(v), g(v')\rangle \in Q].$$

(4) Let P and Q be pers on X. Then define the product per, $P \times Q$, by:

$$\langle\langle v, w\rangle, \langle v', w'\rangle\rangle \in P \times Q \iff \langle v, v'\rangle \in P \text{ and } \langle w, w'\rangle \in Q.$$

(5) Let P be a per on X and $x \in |P|$. Then define

$$[x]_P \stackrel{\text{def}}{=} \{y \in X \mid \langle x, y\rangle \in P\}.$$

(6) Let P be a per on X and $S \subseteq X$. Then define the *restriction of* P *on* S (notation: $P\lceil S$) by:

$$P\lceil S \stackrel{\text{def}}{=} \{\langle u, v\rangle \in P \mid u \in S \text{ and } v \in S\}.$$

In order to interpret types as pers on $\mathbf{D}$, we require the domain of each per corresponding to a type to be a sub-cpo of $\mathbf{D}$.

Definition 12.

(1) Let P be a per on the cpo $\mathbf{D}$. Then P is *complete* iff P satisfies both of the following conditions:

(a) $\langle \bot_{\mathbf{D}}, \bot_{\mathbf{D}}\rangle \in P$;

(b) P is closed under lubs of ω-chains, i.e.

$$\forall i \in \omega.\langle v_i, w_i\rangle \in R \implies \langle \bigsqcup_{i\in\omega} v_i, \bigsqcup_{i\in\omega} w_i\rangle \in P.$$

(2) **CPER** denotes the collection of complete pers (*cpers* for short) on $\mathbf{D}$.

It is easily shown that $\langle \mathbf{CPER}, \subseteq\rangle$ is a complete lattice, hence closed under arbitrary intersections and unions.

The semantic equations for types are as follows:

$$\mathcal{T} : \mathbf{Type} \to \mathbf{CPER}$$

$$\begin{aligned}
\mathcal{T}[\![Bool]\!] &= \{\langle d, d\rangle \mid d \in \mathbf{B}_\bot\};\\
\mathcal{T}[\![Int]\!] &= \{\langle d, d\rangle \mid d \in \mathbf{N}_\bot\};\\
\mathcal{T}[\![\sigma_1 \to \sigma_2]\!] &= \mathcal{T}[\![\sigma_1]\!] \to \mathcal{T}[\![\sigma_2]\!];\\
\mathcal{T}[\![\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}]\!] &= \bigcap_{i=1}^{n} \{\langle q, q'\rangle \mid q, q' \in \mathbf{R} \text{ and } \langle q(l_i), q'(l_i)\rangle \in \mathcal{T}[\![\sigma_i]\!]\};\\
\mathcal{T}[\![[l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n]]\!] &= \bigcup_{i=1}^{n} \{\langle\langle l_i, v\rangle, \langle l_i, v'\rangle\rangle \mid \langle l_i, v\rangle, \langle l_i, v'\rangle \in \mathbf{U} \text{ and } \langle v, v'\rangle \in \mathcal{T}[\![\sigma_i]\!]\}\\
\mathcal{T}[\![\rho r{:}\tau.\{\phi_1, \ldots, \phi_k\}]\!] &= \textit{let } R = \mathcal{T}[\![\tau]\!]\\
&\quad \textit{and } S = \{v \in \mathbf{D} \mid \bigwedge_{j=1}^{k} \mathcal{A}[\![\phi_j]\!]\langle \bot_{\mathbf{EEnv}}, [r \mapsto v]\rangle\}\\
&\quad \textit{in } R\lceil S \textit{ end}.
\end{aligned}$$

Figure 9. The Semantic Equations for Types of μFinal.

We interpret each assertion as an element of non-pointed **T**, since an assertion must be always either *true* or *false* even if evaluation of expressions contained in it would not terminate.

$$\mathcal{A} : \mathbf{Assertion} \to (\mathbf{Env} \to \mathbf{T})$$

$$\mathcal{A}[\![e_1 \leqslant e_2 : \sigma]\!]\varepsilon = (\mathcal{E}[\![e_1]\!]\varepsilon \sqsubseteq \mathcal{E}[\![e_2]\!]\varepsilon);$$
$$\mathcal{A}[\![\textbf{forall } x{:}\sigma.\phi]\!]\varepsilon = \textbf{let } \langle\zeta,\xi\rangle = \varepsilon \textbf{ in } \forall v \in |\mathcal{T}[\![\sigma]\!]|.\mathcal{A}[\![\phi]\!]\langle\zeta[x \mapsto v],\xi\rangle \textbf{ end}.$$

Figure 10. The Semantic Equations for Assertions of μFinal.

We must check the well-definedness of the semantic function $\mathcal{T}$. For this purpose we need a lemma and the proof shows the reason why we have imposed the syntactical constraint (a) to μFinal in Section 3.

Theorem 13 Well-definedness of $\mathcal{T}$. *$\mathcal{T}$ is well-defined. That is, for each $\tau \in$* **Type**,

(1) $$\mathcal{T}[\![\tau]\!] \in \mathbf{CPER};$$

(2) $$\langle wrong, wrong\rangle \notin \mathcal{T}[\![\tau]\!].$$

We now turn to the soundness of our type theory μFINAL with respect to this semantics.

Definition 14. An environment $\varepsilon = \langle\zeta,\mu\rangle$ is said to *respect contexts* Γ,Δ (notation: $\varepsilon \models \Gamma,\Delta$) iff it satisfies both of the following two conditions:

(1) $\zeta \models \Gamma$, i.e. for any variable $x \in \mathrm{dom}(\Gamma)$, $\zeta[\![x]\!] \in |\mathcal{T}[\![\Gamma(x)]\!]|$;
(2) $\xi \models \Delta$, i.e. for any implementation variable $r \in \mathrm{dom}(\Delta)$, $\xi[\![r]\!] \in |\mathcal{T}[\![\Delta(r)]\!]|$.

Finally, we show that the theory μFINAL is sound with respect to this semantics. First, we give some definitions and a lemma.

Definition 15.

(1) Let Σ be a sentence of μFINAL. Then Σ is *satisfied* under an environment $\varepsilon = \langle\zeta,\mu\rangle$ (notation: $\varepsilon \models \Sigma$) iff either one of the following cases holds:
   (a) when $\Sigma \equiv \sigma \leq \tau$,
   $$\mathcal{T}[\![\sigma]\!] \subseteq \mathcal{T}[\![\tau]\!];$$
   (b) when $\Sigma \equiv \Gamma,\Delta \rhd e : \sigma$,
   $$\varepsilon \models \Gamma,\Delta \implies \mathcal{E}[\![e]\!]\varepsilon \in |\mathcal{T}[\![\sigma]\!]|;$$
   (c) when $\Sigma \equiv \Gamma,\Delta \rhd \phi$,
   $$\varepsilon \models \Gamma,\Delta \implies \mathcal{A}[\![\phi]\!]\varepsilon = true.$$

(2) Let Σ be a sentence of μFINAL. Then Σ is *valid* (notation: $\models \Sigma$) iff $\varepsilon \models \Sigma$ for any environment $\varepsilon \in$ **Env**.

Now we can state and prove the soundness theorem for μFINAL.

Theorem 16 Soundness Theorem. *The theory μFINAL is sound with respect to this semantics; i.e. for any sentence Σ of μFINAL*

$$\vdash \Sigma \implies \models \Sigma. \ \blacksquare$$

This soundness theorem is usually presented in forms for special cases (cf. [Cardelli 84]).

Corollary 17 Semantical Soundness Theorem. *If an expression is syntactically typable, then it does not cause any run-time type error. That is*

$$\vdash \Gamma,\Delta \rhd e : \sigma \implies \forall\varepsilon \models \Gamma,\Delta.[\mathcal{E}[\![e]\!]\varepsilon \in |\mathcal{T}[\![\sigma]\!]|].$$

In other words,

$$\vdash \Gamma,\Delta \rhd e : \sigma \implies \forall\varepsilon \models \Gamma,\Delta.[\mathcal{E}[\![e]\!]\varepsilon \neq wrong]. \ \blacksquare$$

Corollary 18 Semantical Subtyping Theorem. *Let σ and τ be types of* μFinal. *Then*

$$\mathcal{T}[\![\sigma]\!] \subseteq \mathcal{T}[\![\tau]\!]. \ \blacksquare$$

6. Discussions, Conclusion and Directions of Future Research

The motivation of our work originates from International Workshop of Semantics of Data Types. Foreword of the proceedings [KMP 84] states that *"The Symposium was intended to bring these somewhat disparate groups together with a view to promoting a common language ...,"* but unfortunately there have been hardly any efforts to integrate logical and algebraic approaches to abstract data types by now. What we have shown in this paper is that the type system with inequational assertions is a natural extension of a typed λ-calculus with record types and the complete partial equivalence model is rich enough to interpret types with inequational assertions.

Intuitively speaking, our notion of type is a collection of values satisfying *at least* some particular properties. Hence it is natural to request that, if each value of an ω-chain satisfies such properties, then the supreme of the chain must also satisfy those properties corresponding to the completeness condition requested to our pers (the pointedness is necessary since we want to have **fix** on all types). On the other hand, when $v_1 \sqsubseteq v_2$, v_1 may lose some information (properties) that v_2 has, hence we have not requested the closedness under approximations like in the class of pers used in [Amadio 91, Cardone 91].

One drawback of our semantics is that the computation of functional applications cannot be performed within a type in general. To be more concrete, let f be a function from type σ to τ and a be a value of σ, then $f(a)$ must be calculated using bases $\langle e_i \rangle_{i \in \omega}$ of a. The point is that some of these bases may not belong to the sub-cpo (the domain of a per) corresponding to the type of a, σ, hence we must perform this calculation in the whole domain $\mathbf{D}$. This is the cost we have paid for our more expressive type system.

Our system can be said a type system combining programming types (usual types of μFun) and specification (inequational assertions as partial correctness requirements) with which we can write specifications for *verification* as well as *executable* programs, hence our μFinal is a good candidate for foundations of type systems of *functional wide-spectrum languages* such as Extended ML [Sannella and Tarlecki 89]. Following extensions to our system are very interesting remaining works:

(1) to give our language typed interpretations;
(2) to strengthen our inequality "$\leqslant$" to "$=$" in assertions;
(3) to extend our system to second-order calculi with polymorphism, existentially quantified types, bounded quantifications, and parameterization of types;
(4) to enrich our system with recursion on types;
(5) to incorporate more sophisticated record calculi such as row variables and selective field updating.

Acknowledgements

The author wish to express his deepest thanks to Professor Dr. Henk Barendregt for his constructive and valuable comments and advices to an earlier version of the present paper and his warm encouragements, and discussions with him have much clarified the present work. Furthermore, the author thanks to Yugo Kashiwagi for his enthusiastic discussions and valuable suggestions.

References

[Amadio 91] Amadio, R. M.: Recursion over Realizability Structures, *Inform. Comput.* **91**, 55–85 (1991).
[Barendregt 81] Barendregt, H. P.: *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam (1981).
[Cardelli 84] Cardelli, L.: A Semantics of Multiple Inheritances, in [KMP 84], 51–67; a revised version appeared in *Inform. Comput.* **76**, 138–164 (1988).
[Cardone 91] Cardone, F.: Recursive Types for Fun, *Theoret. Comput. Sci.* **83**, 29–56 (1991).
[KMP 84] Kahn, G., D. B. MacQueen, and G. Plotkin (eds.): *Semantics of Data Types*, Proceedings of International Symposium, Sophia-Antipolis, June 1984, Lecture Notes in Computer Science **173**, Springer-Verlag, Berlin (1984).
[MTH 90] Milner, R. et al.: *The Definition of Standard ML*, MIT Press, Cambridge MA (1990).
[Plotkin 77] Plotkin, G. D.: LCF Considered as a Programming Language, *Theoret. Comput. Sci.* **5**, 223–255 (1977).
[Plotkin 78] Plotkin, G.: $\mathbf{T}^{\omega}$ as a Universal Domain, *J. Comput. Syst. Sci.* **17**, 209–236 (1978).
[Plotkin 83] Plotkin, G.D.: *Domains*, Advanced Postgraduate Course Notes, Department of Computer Science, University of Edinburgh (1983).
[Reynolds 85] Reynolds, J. C.: Three Approaches to Type Structures, *TAPSOFT-CAAP '85* (H. Ehrig et al. eds.), Lecture Notes in Computer Science **185**, 97–138, Springer-Verlag, Berlin (1985).