Representation independence and data abstraction (preliminary version)

John C. Mitchell

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

One purpose of type checking in programming languages is to guarantee a degree of "representation independence:" programs should not depend on the way stacks are represented, only on the behavior of stacks with respect to push and pop operations. In languages with abstract data type declarations, representation independence should hold for user-defined types as well as built-in types. We study the representation independence properties of a typed functional language (second-order lambda calculus) with polymorphic functions and abstract data type declarations in which data type implementations (packages) may be passed as function parameters and returned as results. The type checking rules of the language guarantee that two data type implementations P and Q are equivalence whenever there is a correspondence between the behavior of the operations of P and the behavior of the operations of Q.

1. Introduction

The second-order (polymorphic) lambda calculus, discovered independently by Girard and Reynolds [Girard 71, Reynolds 74], was proposed by Reynolds as language which captures the essence of type declarations and polymorphic functions. In [Mitchell and Plotkin 85], it was argued that this language could be extended to provide a flexible form of abstract data type declaration. The SOL **abstype** declaration described in [Mitchell and Plotkin 85] is more flexible than the abstract data type declarations provided by many languages in that data type implementations may be passed as parameters and returned as the results of function calls. However, it is not clear *a priori* whether type checking is "secure" in any semantic sense. Furthermore, recent language designs such as Pebble [Burstall and Lampson 84] and Standard ML [MacQueen 85, Milner 85] propose more complex type checking rules that seem more lenient than SOL; it is conceivable that these languages may be less "type secure." The goal of this paper is to understand the semantic properties of type checking in SOL well enough to allow sensible comparisons with other languages. In addition, it is also hoped that this study will be useful in further investigation of related languages. The main technical result is a characterization of when two user-supplied data type implementations are equivalent for all intents and purposes.

In SOL, abstract data type declarations have the form

abstype t **with** $x_1{:}\sigma_1$, ..., $x_k{:}\sigma_k$ **is** M **in** N,

where t is the type name; x_1, ..., x_k are operations on t of types σ_1, ..., σ_k; and M implements the type. The scope of the declaration is the body N. For example, an expression declaring complex numbers looks like

abstype complex **with** create: real→real→complex,
plus: complex→complex→complex,
re: complex→real, im: complex→real
is M
in N,

where N uses complex numbers via the operations *create*, *plus*, *re* and *im*. Abstract data types are implemented by expressions of the form

$$M ::= \textbf{rep}\ \tau\ M_1 \ldots M_k,$$

where τ is a type and M_1, ..., M_k implement operations on that type. (The keyword **rep** is short for *representation*, and is taken from CLU [Liskov et. al. 81].) In SOL, an implementation has a *existential* type $\exists t.\sigma$, which may be thought of as the signature of the data type.

There are three type checking rules for **abstype** expressions. The first, (AB.1) requires that the implementation M match the declaration $x_1{:}\sigma_1$, ..., $x_k{:}\sigma_k$ of operations: M must provide implementations for exactly k operations and the implementation of each operation must have the correct type. The second type checking rule, (AB.2), stipulates that the type name t must not appear in the types of free identifiers in N other than x_1, ..., x_k. This prevents functions other than x_1, ..., x_k from masquerading as operations on the abstract type. The third rule, (AB.3), is that t cannot be free in the type of N. In particular, N cannot have type t, or be one of the operations x_1, ..., x_k. Intuitively, this rule prevents the representation of t from being exported outside the scope of the declaration.

The most controversial rule is (AB.3). This rule is considered briefly in Section 4 of [Reynolds 83] and adopted in Edinburgh ML [Gordon, et. al. 79], but rejected to varying degrees in Pebble [Burstall and Lampson 84], Standard ML [Milner 85, MacQueen 85] and Martin-Löf's constructive type theory [Martin-Löf 79]. Some direct objections to (AB.3) are described in [MacQueen 86]. However, we cannot drop (AB.3) from SOL without drastically changing the type expression of the language. As it stands, type expressions are separated from the ordinary terms (λ-expressions) since terms do not appear in type expressions. But the type of the expression

abstype t **with** x:σ **is** M **in** x

depends on the value of M (i.e., the representation type supplied by M), not just the type of M. Therefore, in the case where M might be a formal parameter, we can only write the type of

abstype t **with** x:σ **is** M **in** x

as a type expression involving M. In addition, the representation independence properties demonstrated in Theorems 6 and 7 rely on (AB.3). This is not to say that representation independence fails for languages without (AB.3), only that the representation independence properties are likely to be more difficult to describe.

Intuitively, the purpose of a representation independence theorem is to show that certain implementation decisions do not effect the meanings of programs. Various representation independence, or "abstraction," theorems have been proposed by Reynolds, Donahue and Haynes [Donahue 79, Haynes 84, Reynolds 74, Reynolds 83]. Essentially, all of these theorems are slight generalizations of the statement,

> If two interpretations $\mathfrak{A}$ and $\mathfrak{B}$ are related in a certain way, then the meaning $\mathfrak{A}[\![M]\!]$ of any closed term M in $\mathfrak{A}$ is related to the meaning $\mathfrak{B}[\![M]\!]$ of M in $\mathfrak{B}$ in the same certain way.

The pragmatic consequence of this sort of theorem is that if two programming language interpreters are related in this "certain way," then the result of executing any program using one interpreter will correspond to the result of executing the same program using the other interpreter. Thus the precise statement of the theorem describes the kind of implementation decisions that do not effect the meanings of programs. While the representation independence theorems for second-order lambda calculus without **abstype** proposed in [Donahue 79, Reynolds 74, Reynolds 83] have some shortcomings[1], a general representation independence theorem is proved in [Mitchell and Meyer 85] (using the model theory developed in [Bruce, Meyer and Mitchell 85, Bruce and Meyer 84, Mitchell 84c]).

In languages with abstract data type declarations, implementation decisions may be made by

1. A critique of the results of [Donahue 79, Reynolds 74, Reynolds 83] appears in [Haynes 84]. The representation independence theorem of [Haynes 84] repairs some of the shortcomings of previous

programmers, and so representation independence becomes a programmer concern as well as a language implementation issue. Since a type such as `symbol_table` may be implemented once at the beginning of a development project, and then optimized later, it is useful to know how changes in the implementation will effect the behavior of procedures that use `symbol_table`. In particular, it is important to know that certain changes do not effect the behavior of other procedures. The theorem in Section 6 of this paper demonstrates that the *observable behavior* of a program is not effected by certain changes in the definitions of abstract data types. The main ideas are described by example in the following Section.

2. An Illustrative Example: Integer Multisets

2.1. Introduction

We look at an example data type, integer multisets, assuming that we observe the behavior of programs by computing integers. Thus multisets are used only at intermediate stages in computation, and we only care about the behavior of multisets insofar as we can observe them by producing integer results. In Section 2.2, we compare implementations of integer multisets as if the implementations were provided as part of the semantics of a simple programming language. In Section 2.3, we look at alternate implementations defined in the language itself, deriving a definition and characterization of observable equivalence. To keep the semantics simple, we will assume that programs are first-order terms interpreted over multi-sorted first-order structures, returning to the more complicated second-order lambda calculus in Section 3.

2.2. Built-in Types

An interpretation for expressions involving integers and integer multisets consists of the set of integers with 0, 1 and +, and a set s of multisets with operations

empty: s, insert: int→s→s, count: s→int

Informally, *empty* is the empty multiset, *insert* adds an integer to a multiset, and *count* returns the multiplicity of an element of a multiset. Let us consider two structures

$$\mathfrak{A} = \langle \mathbb{N}, s^{\mathfrak{A}}, 0, 1, +, \text{empty}^{\mathfrak{A}}, \text{insert}^{\mathfrak{A}}, \text{count}^{\mathfrak{A}} \rangle$$

$$\mathfrak{B} = \langle \mathbb{N}, s^{\mathfrak{B}}, 0, 1, +, \text{empty}^{\mathfrak{B}}, \text{insert}^{\mathfrak{B}}, \text{count}^{\mathfrak{B}} \rangle$$

that both interpret the integers and 0, 1, + in the usual standard way. The set $s^{\mathfrak{A}}$ of multisets in $\mathfrak{A}$ may be different from the set $s^{\mathfrak{B}}$ of multisets of $\mathfrak{B}$, and of course the multiset operations may be different.

We say that structures $\mathfrak{A}$, $\mathfrak{B}$ are *observationally equivalent with respect to the integers* if, for any closed integer term M, we have

$$\mathfrak{A}[\![M]\!] = \mathfrak{B}[\![M]\!],$$

i.e. the meaning of M in $\mathfrak{A}$ is the same as the meaning of M in $\mathfrak{B}$. Under what conditions will $\mathfrak{A}$ and $\mathfrak{B}$ be observationally equivalent?

A first guess is that $\mathfrak{A}$ and $\mathfrak{B}$ are observationally equivalent iff there exists some kind of mapping between $\mathfrak{A}$ and $\mathfrak{B}$, say a homomorphism. This is partly correct.

LEMMA 1. *If there is a homomorphism $h:\mathfrak{A}\to\mathfrak{B}$, then $\mathfrak{A}$ and $\mathfrak{B}$ are observationally equivalent.*
The proof of this lemma is quite straightforward. Since h(0)=0, h(1)=1 and h(x+y)=h(x)+h(y), it is easy to see that h must be the identity on $\mathbb{N}$. By induction on terms, we can verify that for any closed first-order term M, we have $h(\mathfrak{A}[\![M]\!]) = \mathfrak{B}[\![M]\!]$.

It seems worthwhile to discuss two reason why the converse of this lemma fails. The first has to do with the fact that elements of $s^{\mathfrak{A}}$ and $s^{\mathfrak{B}}$ which are not definable by terms are irrelevant. For example, suppose $\mathfrak{A}$ is derived from some implementation $\mathfrak{B}$ of integer multisets by adding some "nonstandard" multiset a with the property

theorems, but the theorem is stated in an elaborate way and it only applies to a special class of models.

insert x a = a for all x.

Then $\mathfrak{A}$ and $\mathfrak{B}$ will be observationally equivalent, since the multiset a will never occur in practice, but there is no homomorphism h from $\mathfrak{A}$ to $\mathfrak{B}$ since there is no reasonable choice for h(a).

Another reason why the converse of the lemma fails may be explained in programming language terms. One way of representing a multiset is as a linked list of pairs of the form

<element, count>,

where the integer *count* is the number of times *element* has been inserted. The *empty* multiset is then the empty linked list, *insert* adds a new pair or increments the appropriate count, and the function *count* searches the list and returns the appropriate count. An alternative representation makes sense if we assume that the integers 1, ..., 10 will occur quite frequently, with other numbers much less likely. In this case, we might use an array of length 10 to count the number of times 1, ..., 10 are inserted, together with a simple list of other insertions in the order they occur. Note that repeatedly inserting 12, for example, will result in 12 appearing several times in the list. Provided that *insert* and *count* are implemented properly, these two representations will be observationally equivalent. However, there can be no homomorphism from the first to the second. To see why this is so, consider the result of inserting three elements into the empty multiset. We use the abbreviation

$insert^3$ x y z for *insert* x (*insert* y (*insert* z *empty*)).

In the first representation, assuming x, y>10, we have

$insert^3$ x y x = $insert^3$ x x y = <<x, 2>, <y, 1>>

whereas in the second representation.

$insert^3$ x y x = Array;<x, y, x> ≠ Array;<x, x, y> = $insert^3$ x x y.

Any homomorphism h from the first representation to the second would have to map the list of pairs <<x, 2>, <y, 1>> to both <x, y, x> and <x, x, y>, but this is impossible. Conversely,

$insert^3$ 1 2 3 = $insert^3$ 3 2 1

in the second representation, since both set Array[1]=Array[2]=Array[3]=1. However, in the first representation using lists of <element, counts> pairs, $insert^3$ 1 2 3 and $insert^3$ 3 2 1 yield lists in different orders.

In the example above, both implementations are homomorphic images of some "initial" implementation, but there is no structure preserving function from one to the other. The correct correspondence between observationally equivalent implementations involves relations.

LEMMA 2. *Structures $\mathfrak{A}$ and $\mathfrak{B}$ for integer multisets are observationally equivalent iff there is a relation $R^s \subseteq s^{\mathfrak{A}} \times s^{\mathfrak{B}}$ such that*

$R^s(empty^{\mathfrak{A}}, empty^{\mathfrak{B}})$,

$R^s(a, b) \supset R^s(insert^{\mathfrak{A}}\ x\ a, insert^{\mathfrak{B}}\ x\ b)$,

$R^s(a, b) \supset count^{\mathfrak{A}}\ x\ a = count^{\mathfrak{B}}\ x\ b$

Intuitively, the relation R^s specifies, for each multiset $a \in s^{\mathfrak{A}}$, the collection $\{b \in s^{\mathfrak{B}} \mid R^s(a, b)\}$ of all "behaviorally equivalent" multisets.

We can state this lemma a little more generally by introducing "logical relations;" this definition is based on relations used in [Friedman 75, Plotkin 80, Statman 82, Tait 67]. Let $\mathfrak{A}$ and $\mathfrak{B}$ be multi-sorted first-order structures for sorts $s_1, \ldots, s_j$ and functions (or constants) $f_1, \ldots, f_k$, i.e.,

$\mathfrak{A} = <s_1^{\mathfrak{A}}, \ldots, s_j^{\mathfrak{A}}, f_1^{\mathfrak{A}}, \ldots, f_k^{\mathfrak{A}}>$

$\mathfrak{B} = <s_1^{\mathfrak{B}}, \ldots, s_j^{\mathfrak{B}}, f_1^{\mathfrak{B}}, \ldots, f_k^{\mathfrak{B}}>$.

A *first-order logical relation $\mathfrak{R}$ over $\mathfrak{A}$, $\mathfrak{B}$* is a family of relations $\mathfrak{R}=\{R^s \mid 1 \le i \le j\}$ such that

$R^{s_i} \subseteq s_i^{\mathfrak{A}} \times s_i^{\mathfrak{B}}$, and

for $f: t_1 \to \ldots \to t_{n+1}$, we have $R^{t_i}(f^{\mathfrak{A}}(x_1, \ldots, x_n), f^{\mathfrak{B}}(y_1, \ldots, y_n))$ whenever $R^{t_i}(x_i, y_i)$ $i \le n$.

Less formally, $\mathcal{R}$ is a logical relation on $\mathfrak{A}$ and $\mathfrak{B}$ if $\mathcal{R}$ relates the sorts of $\mathfrak{A}$ and $\mathfrak{B}$ in such a way that every function f interpreted by $\mathfrak{A}$ and $\mathfrak{B}$ maps related arguments to related results.

One important property of logical relations is that the meanings of expressions are related.

LEMMA 3. *Let $\mathcal{R}$ be a first-order logical relation over $\mathfrak{A}$ and $\mathfrak{B}$ and let η_a, η_b be environments such that $R(\eta_a(x), \eta_b(x))$ for every variable x. If M is any first-order term, then $R(\mathfrak{A}[\![M]\!]\eta_a, \mathfrak{B}[\![M]\!]\eta_b)$.*

Using logical relations, Lemma 2 can be restated

Structures $\mathfrak{A}$ and $\mathfrak{B}$ for integer multisets are observationally equivalent iff there is a logical relation $\mathcal{R}=\{R^{int}, R^s\}$ on $\mathfrak{A}$, $\mathfrak{B}$ such that R^{int} is the identity relation on $\mathbb{N}$.

These properties of logical relations are proved for second-order lambda calculus in [Mitchell and Meyer 85].

2.3. User-defined Types

We now extend the syntax of terms to include abstract data type declarations

abstype t **with** $x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k$ **is** M **in** N,

where a representation M has the form

M ::= **rep** τ $M_1 \ldots M_k$.

In a representation M, the type τ is used as the carrier of the type being defined, while $M_1, \ldots, M_k$ implement operations $x_1, \ldots, x_k$. As mentioned in the Introduction, t cannot appear in the type of N and the type of M_i must match the type σ_i of the declared operation x_i (this means that M_i must have type $[\tau/t]\sigma_i$). Since our example language only allows first-order terms, we will implement operations by interpreting first-order terms $M_1, \ldots, M_k$ with free variables as function expressions.

Since we observe implementations P and Q for integer multisets using integer expressions, it seems natural to consider P and Q observationally equivalent if any integer term M involving P always has the same meaning as the term we obtain by replacing each occurrence of P by Q. We can simplify and generalize this notion of observational equivalence using logical relations. If $\mathcal{R}$ is a logical relation over $\mathfrak{A}$ and $\mathfrak{B}$, with R^{int} the identity, then we know P and Q will be observationally equivalent if, for every N, the meaning of

abstype s **with** empty: s, insert: int→s→s, count: s→int **is** P **in** N

is related to the meaning of the term

abstype s **with** empty: s, insert: int→s→s, count: s→int **is** Q **in** N.

This motivates the definition of observational equivalence with respect to a logical relation.

A *multiset context* is an expression $\mathcal{C}[*]$

abstype s **with** empty: s, insert: int→s→s, count: s→int **is** * **in** N

with a place * to insert an implementation of integer multisets. If P and Q are implementations of integer multisets, and $\mathcal{R}$ is a logical relation over $\mathfrak{A}$ and $\mathfrak{B}$, we say P and Q are *observationally equivalent with respect to* $\mathcal{R}$ if $\mathcal{R}(\mathfrak{A}[\![\mathcal{C}[P]]\!], \mathfrak{B}[\![\mathcal{C}[Q]]\!])$ for every multiset context $\mathcal{C}[*]$.

If P and Q have the form

P ::= **rep** s_p P_1 P_2 P_3 and Q ::= **rep** s_q Q_1 Q_2 Q_3,

a sufficient condition for observational equivalence is that s_p and s_q are the same, so that we already have a relation $R^{s_p}=R^{s_q}$ in $\mathcal{R}$, and that the meaning of each P_i is related to the corresponding Q_i. However, this condition is clearly not necessary. For example, even if $s_p=s_q=$int, there are many ways of representing integer multisets as integers. So the empty multiset P_1 used in P need not be the same as the empty multiset Q_1 used in Q.

A better test for observational equivalence of data type implementations is obtained by

considering extended relations over extended structures. The meanings of **abstype** expressions involving P and Q will have related values if there exists an additional relation $R^s \subseteq s_p^{\mathfrak{A}} \times s_q^{\mathfrak{B}}$ so that $\mathcal{R} \cup \{R^s\}$ is a logical relation over two extended structures $\mathfrak{A}^+$ and $\mathfrak{B}^+$. The extended structures are defined by

$$\mathfrak{A}^+ = \langle s_1^{\mathfrak{A}}, \ldots, s_j^{\mathfrak{A}}, s_p^{\mathfrak{A}}, f_1^{\mathfrak{A}}, \ldots, f_k^{\mathfrak{A}}, \mathfrak{A}[\![P_1]\!], \mathfrak{A}[\![P_2]\!], \mathfrak{A}[\![P_3]\!] \rangle$$

$$\mathfrak{B}^+ = \langle s_1^{\mathfrak{B}}, \ldots, s_j^{\mathfrak{B}}, s_q^{\mathfrak{B}}, f_1^{\mathfrak{B}}, \ldots, f_k^{\mathfrak{B}}, \mathfrak{B}[\![Q_1]\!], \mathfrak{B}[\![Q_2]\!], \mathfrak{B}[\![Q_3]\!] \rangle.$$

Since the structure $\mathfrak{A}^+$ has a second "copy" of s_p and $\mathfrak{B}^+$ a second copy of s_q, we can use a new relation between s_p and s_q to give a correspondence between multisets in P and multisets in Q.

LEMMA 4. *Implementations P and Q for integer multisets are observationally equivalent with respect to relation $\mathcal{R}$ over $\mathfrak{A}$ and $\mathfrak{B}$ iff there exists a relation $R^s \subseteq s_p^{\mathfrak{A}} \times s_q^{\mathfrak{B}}$ such that $\mathcal{R} \cup \{R^s\}$ is a logical relation over $\mathfrak{A}^+$ and $\mathfrak{B}^+$.*

This Lemma shows that two implementation P and Q are observationally equivalent iff there is a correspondence between the operations of P and the operations of Q.

In second-order lambda calculus, implementations P and Q will be terms of the language. Both P and Q have type

$$\exists s.\ s \wedge \text{int} \to s \to s \wedge s \to \text{int},$$

which is just the signature of integer multisets with the operation names left out. A logical relation over second-order models will include a relation for each type, and hence a relation between data type implementations -- elements of existential types. We extend the definition of second-order logical relation given in [Mitchell and Meyer 85] to existential types by relating all pairs of observationally equivalent implementations. Theorem 7 in Section 6 generalizes Lemma 4 above by showing that implementations P and Q are related by some second-order logical relation iff there is a correspondence between the operations of P and the operations of Q. Sections 3 through 5 present the necessary preliminaries.

3. Syntax

Second-order lambda calculus (SOL, or $\mathcal{SA}$) is an extension of the ordinary typed lambda calculus. In addition to allowing abstraction with respect to typed variables, the second-order system allows abstraction with respect to types themselves. We use a version of the language in which every term has a type and every subexpression of a type expression has a kind. (Kinds were first used in [McCracken 79].) The subexpressions of type expressions, which may be type expressions or operators like $\to$ and $\forall$, will be called constructors. We define the sets of kinds and constructors before introducing the syntax and type checking rules for terms.

We use the constant T to denote the kind consisting of all types. The set of kind expressions is given by the grammar

$$\kappa ::= T \mid \kappa_1 \Rightarrow \kappa_2.$$

The kind expression $\kappa_1 \Rightarrow \kappa_2$ will be interpreted as a set of functions from κ_1 to κ_2. For example, functions from types to types will have kind $T \Rightarrow T$. We define the set of constructor expressions, beginning with a set of constructor constants. Let $\mathcal{C}_{cst}$ be a set of constant symbols c^κ, each with a specified kind (which we write as a superscript when necessary) and let $\mathcal{V}_{cst}$ be a set of variables v^κ, each with a specified kind. We assume we have infinitely many variables of each kind.

The constructor expressions over $\mathcal{C}_{cst}$ and $\mathcal{V}_{cst}$, and their kinds, are defined by the following derivation system

$$c^\kappa : \kappa, \quad v^\kappa : \kappa$$

$$\frac{\mu : \kappa_1 \Rightarrow \kappa_2, \quad \nu : \kappa_1}{\mu\nu : \kappa_2}$$

$$\frac{\mu:\kappa_2}{\lambda v^{\kappa_1}.\mu : \kappa_1 \Rightarrow \kappa_2}$$

For example $(\lambda v^T.v^T)c^T$ is a constructor expression with kind T. A special class of constructor expressions are the type expressions, the constructor expressions of kind T. Since we will often be concerned with type expressions rather than arbitrary constructor expressions, it will be useful to distinguish them by notational conventions. We adopt the conventions that

r, s, t, ... denote type variables

ρ, σ, τ, ... denote type expressions.

As in the definition above, we will generally use μ and ν for constructor expressions. We include the usual second-order types in the language by assuming that $\mathscr{C}_{cst}$ contains the function-type constructor constant

$$\rightarrow : T \Rightarrow (T \Rightarrow T)$$

and the polymorphic type and "data type" type constructor constants

$$\forall, \exists : (T \Rightarrow T) \Rightarrow T.$$

As usual, we write $\rightarrow$ as an infix operator, as in the type expression $\sigma \rightarrow \tau$, and write $\forall t.\sigma$ for $\forall(\lambda t.\sigma)$. The advantage of working in a language with constructor expressions is that we may extend the language to include products or sums by adding the appropriate constants to $\mathscr{C}_{cst}$. We write $\vdash_c \mu = \nu$ if the equation $\mu = \nu$ follows from the usual axioms and rules of inference for typed lambda expressions (or, equivalently, if μ and ν are α, β, η-interconvertible.)

As in most typed programming languages, the type of an $\mathcal{GA}$ term will depend on the context in which it occurs. We must know the types of all free variables before we can assign a lambda expression a type. Let $\mathscr{V}_{term}$ be an infinite collection of variables. A *syntactic type assignment* B is a function from a subset (finite or infinite) of $\mathscr{V}_{term}$ to type expressions. For any syntactic type assignment B, let B[x:γ] be the type assignment which is identical to B except that (B[x:γ])(x)=γ.

Let B be a syntactic type assignment, and let $\mathscr{C}_{term}$ be a set of constants, each with a specified closed type. We define terms and their types using derivation rules for formulas B⊢M:γ (read "M has type γ with respect to B"). If M is a term and t does not occur free in B(x) for any x free in M, then t is *bindable in* M *with respect to* B. We use $\{\tau/t\}\sigma$ to denote the result of substituting τ for free occurrences of t in σ. Constants and variables are typed as follows.

$B \vdash c^{\tau} : \tau$ and $B \vdash x : B(x)$ if x is in the domain of B

The typing rules for compound terms are

$$(\rightarrow E)\ \frac{B \vdash M : \sigma \rightarrow \tau,\ B \vdash N : \sigma}{B \vdash MN : \tau}$$

$$(\rightarrow I)\ \frac{B[x:\sigma] \vdash M : \tau}{B \vdash \lambda x:\sigma.M : \sigma \rightarrow \tau}$$

$$(\forall E)\ \frac{B \vdash M : \forall t.\sigma}{B \vdash M\tau : \{\tau/t\}\sigma}$$

$$(\forall I)\ \frac{B \vdash M : \tau}{B \vdash \lambda t.M : \forall t.\tau}\quad \text{t bindable in M w.r.t. B}$$

$$(\exists\ E)\quad \frac{B\vdash M:\exists t.\sigma,\ B\vdash N:\rho}{B\vdash \textbf{abstype } t \textbf{ with } x:\sigma \textbf{ is } M \textbf{ in } N:\ \rho}\quad t \text{ not free in } \rho \text{ or } B(y) \text{ for } y\neq x$$

$$(\exists\ I)\quad \frac{B\vdash M:[\tau/t]\sigma}{B\vdash \textbf{rep}_{\exists t.\sigma}\ \tau\ M:\ \exists t.\sigma}$$

$$(\text{type eq})\quad \frac{\vdash_c \gamma=\rho,\ B\vdash M:\gamma}{B\vdash M:\rho}$$

The language $\mathcal{SL}_B(\mathcal{C}_{cst}, \mathcal{C}_{term})$ is the set of terms M over constructor constants $\mathcal{C}_{cst}$ and term constants $\mathcal{C}_{term}$ such that $B\vdash M:\sigma$ for some σ. We often write $Type_B(M)$ for any σ such that $B\vdash M:\sigma$. While we have only allowed one operation in **rep** τ M, there is no loss of generality since M may be a tuple of operations. (See [Mitchell and Plotkin 85]; pairing may be "simulated" in the second-order lamda calculus above as described in [Bruce, Meyer and Mitchell 85].)

4. Models

4.1. Kind Structures

The semantics of constructor expressions are the familiar semantics of the simply typed lambda calculus.

A *kind structure* $\mathcal{K}ind$ for a set $\mathcal{C}_{cst}$ of constructor constants is a tuple

$$\mathcal{K}ind = \langle\{Kind^\kappa\},\{\Phi_{\kappa_1\Rightarrow\kappa_2}\},\mathcal{I}\rangle,$$

where $\{Kind^\kappa\}$ is a family of sets indexed by kinds κ, $\{\Phi_{\kappa_1\Rightarrow\kappa_2}\}$ is a family of one to one and onto functions indexed by kind expressions κ_1 and κ_2 such that

$$\Phi_{\kappa_1\Rightarrow\kappa_2} : Kind^{\kappa_1\Rightarrow\kappa_2} \to [Kind^{\kappa_1} \to Kind^{\kappa_2}]$$

for $[Kind^{\kappa_1} \to Kind^{\kappa_2}]$ a collection of functions from $Kind^{\kappa_1}$ to $Kind^{\kappa_2}$, and $\mathcal{I}:\mathcal{C}_{cst} \to \cup_\kappa Kind^\kappa$ such that $\mathcal{I}$ preserves kinds, i.e. $\mathcal{I}(c^\kappa)\in Kind^\kappa$. Since constructor expressions include all typed lambda expressions, $\mathcal{K}ind$ must be a model of the simple typed lambda calculus.

4.2. Frames and Environment Models

Models are defined by first describing a structure called a frame, and then distinguishing models from arbitrary frames. A *second order frame* $\mathcal{F}$ *for* $\mathcal{SL}(\mathcal{C}_{cst}, \mathcal{C}_{term})$ is a tuple

$$\mathcal{F} = \langle \mathcal{K}ind,\ \mathcal{D}om,\ \{\Phi_{a,b} \mid a,b\in Kind^T\},\ \{\Phi_f \mid f\in Kind^{T\Rightarrow T}\}\rangle$$

satisfying conditions (i) through (iv) below.

(i) $\mathcal{K}ind = \langle\{Kind^\kappa\},\ \{\Phi_{\kappa_1,\ \kappa_2}\},\ \mathcal{I}\rangle$ is a kind structure for $\mathcal{C}_{cst}$

(ii) $\mathcal{D}om = \langle\{Dom^a \mid a\in Kind^T\},\ \mathcal{I}_{\mathcal{D}om}\rangle$ with each Dom^a a set, and

$$\mathcal{I}_{\mathcal{D}om} : \mathcal{C}_{term} \to \cup_a Dom^a \text{ satisfies } \mathcal{I}_{\mathcal{D}om}(c^\tau) \in Dom^{[\![\tau]\!]} \text{ for all } c^\tau \text{ in } \mathcal{C}_{term},$$

(iii) For each $a,b\in Kind^T$, we have a set $[Dom^a \to Dom^b]$ of functions from

Dom^a to Dom^b with $\Phi_{a,b} : Dom^{a\to b} \to [Dom^a \to Dom^b]$ a bijection.

(iv) For every $f \in Kind^{[T\Rightarrow T]}$, we have a subset $[\Pi_{a\in Kind^T} Dom^{f(a)}] \subseteq \Pi_{a\in Kind^T} Dom^{f(a)}$

with $\Phi_f : Dom^{\forall f} \to [\Pi_{a\in Kind^T} Dom^{f(a)}]$ a bijection.

(v) For every $f \in Kind^{[T \to T]}$ and a, $b \in Kind^T$, we have mappings

$$Sum_{a,f}: Dom^{\forall t.ft \to a} \to Dom^{\exists f \to a} \text{ and } Inj_{b,f}: Dom^{fb} \to Dom^{\exists f}$$

with $\Phi_{\exists f,a}(Sum_{a,f}\, g)(Inj_{b,f}\, d) = \Phi_{fb,a}(\Phi_{\lambda t.ft \to a}\, g)\, b)d$

Essentially, condition (iii) states that $Dom^{a \to b}$ must "represent" some set $[Dom^a \to Dom^b]$ of functions from Dom^a to Dom^b. Similarly, condition (iv) specifies that $Dom^{\forall f}$ must represent some subset $[\Pi_{a \in T} Dom^{f(a)}]$ of the product $\Pi_{a \in T} Dom^{f(a)}$. Some intuition for condition (v) may be gained by comparing $\exists f$ to an infinite sum (see [Mitchell and Plotkin 85]).

Terms are interpreted using Φ for application, Φ^{-1} for abstraction, **Sum** for **abstype** and **Inj** for **rep**. Since different Φ and Φ^{-1} functions are used, depending on the types of terms, the meaning of a term M in frame $\mathcal{F}$ will be defined relative to some type assignment B. We will also need to assume that our environments map variables to elements of the correct types. If B is a type assignment and η an environment mapping $\mathcal{V}_{cst}$ to elements of the appropriate kinds, and $\mathcal{V}_{term}$ to elements of $\cup\mathcal{D}om$, we say that η *satisfies* B, written $\eta \models B$, if

$$\eta(x) \in [\![B(x)]\!]\eta$$

for each variable $x \in dom(B)$.

Let $\mathcal{F}$ be a second-order frame and let B be a syntactic type assignment. If $\eta \models B$, then the meanings of terms of $\mathcal{G}\Lambda_B$ are defined inductively as follows:

$$[\![x]\!]^B\eta = \eta(x),$$

$$[\![c]\!]^B\eta = \mathcal{G}_{\mathcal{D}om}(c),$$

$$[\![MN]\!]^B\eta = (\Phi_{a,b}[\![M]\!]^B\eta)\, [\![N]\!]^B\eta,$$

where $Type_B(M) = \sigma \to \tau$ and $a = [\![\sigma]\!]\eta$, $b = [\![\tau]\!]\eta$,

$$[\![\lambda x{:}\sigma.M]\!]^B\eta = \Phi^{-1}_{a,b}\, g, \text{ where}$$

$g(d) = [\![M]\!]^{B[x:\sigma]}\eta[d/x]$ for all $d \in Dom^a$ and

a, b are the meanings of σ and $Type_B(M)$ in η

$$[\![M\tau]\!]^B\eta = (\Phi_f\, [\![M]\!]^B\eta)\, [\![\tau]\!]\eta\,,$$

where $Type_B(M) = \forall t.\sigma$ and $f = [\![\lambda t.\sigma]\!]\eta$,

$$[\![\lambda t.M]\!]^B\eta = \Phi^{-1}{}_f\, g, \text{ where}$$

for all $a \in Kind^T$, $g(a) = [\![M]\!]^{B'}\eta[a/t]$ for $B' = B|_{FV(M)}$ and

$f \in Kind^{T \Rightarrow T}$ is the function $[\![\lambda t.Type_B(M)]\!]\eta$

$$[\![\textbf{abstype } t \textbf{ with } x{:}\sigma \textbf{ is } M \textbf{ in } N]\!]^B\eta =$$

$Sum_{f,b}([\![\lambda t.\lambda x{:}\sigma.N]\!]^B\eta)\, [\![M]\!]\eta$, where

$f \in Kind^{[T \to T]} = [\![\lambda t.\sigma \to Type_B(N)]\!]\eta$ and $b = [\![Type_B(N)]\!]\eta$

$$[\![\textbf{rep}_{\exists t.\sigma}\, \tau\, M]\!]^B\eta = Inj_{f,[\![t]\!]\eta}\, [\![M]\!]^B\eta, \text{ where}$$

$f \in Kind^{[T \to T]}$ is the function $[\![\lambda t.\sigma]\!]^B\eta$.

In these inductive clauses, there is no guarantee that $[\![M]\!]^B\eta$ is defined for all M, since g in the $\lambda x{:}\sigma.M$ case may not be in the domain of $\Phi^{-1}_{a,b}$, and similarly for g in the $\lambda t.M$ case. Therefore, we make the

following definition. An *environment model* for the second-order lambda calculus is a second-order frame such that for all $\eta \models B$ and all terms M of $\mathcal{SA}_B$, the meaning $[\![M]\!]^B\eta$, as defined above, exists. When there is no danger of confusion we leave off the superscript B on the meaning of terms and simply write $[\![M]\!]\eta$. It is easy to check that the meanings of terms have the appropriate semantic types. See [Bruce and Meyer 84, Bruce, Meyer and Mitchell 85, Mitchell 84c] for details.

5. Logical Relations

In studying representation independence, we use a logical relation $\mathcal{R}$ over models $\mathcal{A}$ and $\mathcal{B}$ to establish an observational equivalence between $\mathcal{A}$ and $\mathcal{B}$. If $\mathcal{R}(a, b)$, then we think of b as equivalent in $\mathcal{B}$ to a in $\mathcal{A}$. The essential property of a logical relation $\mathcal{R}$ on functions f in $\mathcal{A}$ and g in $\mathcal{B}$ of appropriate types is that $\mathcal{R}(f, g)$ holds iff $\mathcal{R}(f(a), g(b))$ for all related a, b of appropriate types. In the case that f and g are polymorphic functions, $\mathcal{R}(f, g)$ is determined by whether $\mathcal{R}(f(a), g(b))$ for all related types a from $\mathcal{A}$ and b from $\mathcal{B}$. We define $\mathcal{R}(a, b)$ for elements a and b of corresponding existential types according to whether a and b are observationally equivalent with respect to $\mathcal{R}$ (see Section 2.3 for motivation). Like the definition of $\mathcal{R}(f, g)$ for polymorphic functions f and g, the definition of $\mathcal{R}(a, b)$ may seem circular. However, we are able to give a more concrete description of $\mathcal{R}(a, b)$ in Section 6. Our description in Section 6 also shows that if $a = \mathbf{rep}\ a_0\ a_1$ and $b = \mathbf{rep}\ b_0\ b_1$, then $\mathcal{R}(a, b)$ depends only on the correspondence between the behavior of operation a_1 on type a_0 and the behavior of operation b_1 on type b_0.

5.1. $\mathcal{TA}$ Logical Relations

Logical relations over second-order models involve ordinary logical relations over kind structures. Let $\mathcal{K}ind_1$ and $\mathcal{K}ind_2$ be two kind structures for $\mathcal{C}_{\text{kind}}$. A $\mathcal{TA}$ *logical relation* $\mathcal{R}$ over $\mathcal{K}ind_1$ and $\mathcal{K}ind_2$ is a family $\{R^\kappa\}$ of relations such that

$$R^\kappa \subseteq \text{Kind}_1{}^\kappa \times \text{Kind}_2{}^\kappa \text{ for each kind } \kappa$$

$$\langle s, t\rangle \in R^{\kappa_1 \to \kappa_2} \text{ iff } \forall s_1 \in \kappa^{\mathcal{S}}.\forall t_1 \in \kappa^{\mathcal{T}}{}_{,1} \langle s_1, t_1\rangle \in R^{\kappa_1} \supset \langle ss_1, tt_1\rangle \in R^{\kappa_2}$$

We say that $\mathcal{R}$ *preserves* a constant c of kind κ if

$$\langle \mathcal{S}_1(c), \mathcal{S}_2(c)\rangle \in R^\kappa$$

and generally assume that a relation $\mathcal{R}$ preserves all constants from $\mathcal{C}_{\text{kind}}$. We will always assume that any logical relation preserves $\to$, $\forall$, $\exists \in \mathcal{C}_{\text{kind}}$. The basic property of logical relations, called the Fundamental Theorem of Logical Relations in [Statman 82], is

THEOREM 5. *(Fundamental Theorem of $\mathcal{TA}$ Logical Relations [Statman 82]) Let $\mathcal{R}$ be a constant-preserving logical relation over kind structures $\mathcal{K}ind_1$, $\mathcal{K}ind_2$ and let η_s, η_t be environments for $\mathcal{K}ind_1$ and $\mathcal{K}ind_2$ such that for each variable v of kind κ we have $\langle \eta_s(v), \eta_t(v)\rangle \in R^\kappa$. Then for any constructional μ of kind κ,*

$$\langle \mathcal{K}ind_1[\![\mu]\!]\eta_s, \mathcal{K}ind_2[\![\mu]\!]\eta_t\rangle \in R^\kappa.$$

5.2. $\mathcal{SA}$ Logical Relations

A logical relation $\mathcal{R}$ over models $\mathcal{A}$ and $\mathcal{B}$ will consist of a $\mathcal{TA}$ logical relation over $\mathcal{K}ind_{\mathcal{A}}$ and $\mathcal{K}ind_{\mathcal{B}}$, together with a family of relations over $\mathcal{D}om_{\mathcal{A}}$ and $\mathcal{D}om_{\mathcal{B}}$. The relations on $\mathcal{D}om_{\mathcal{A}}$ and $\mathcal{D}om_{\mathcal{B}}$ will be indexed by pairs of related types $a \in \text{Kind}_{\mathcal{A}}{}^T$, $b \in \text{Kind}_{\mathcal{B}}{}^T$. More precisely, let $\mathcal{A}$ and $\mathcal{B}$ be models for $\mathcal{SA}(\mathcal{C}_{\text{cst}}, \mathcal{C}_{\text{term}})$. An $\mathcal{SA}$-*logical relation* over $\mathcal{A}$, $\mathcal{B}$ is a family $\mathcal{R}$ of relations

$$R^\kappa \subseteq \text{Kind}_{\mathcal{A}}{}^\kappa \times \text{Kind}_{\mathcal{B}}{}^\kappa \text{ for each kind } \kappa$$

$$R^{a,b} \subseteq \text{Dom}_{\mathcal{A}}{}^a \times \text{Dom}_{\mathcal{B}}{}^b \text{ for each } \langle a, b\rangle \in R^T$$

such that

(LR.1) $\{R^\tau\}$ is a $\mathcal{TA}$-logical relation over $\mathcal{K}ind_{\mathcal{U}}$, $\mathcal{K}ind_{\mathcal{B}}$

(LR.2) For all $<a_1, b_1>$, $<a_2, b_2> \in R^T$ and $<c, d> \in Dom_{\mathcal{U}}^{a_1 \to a_2} \times Dom_{\mathcal{B}}^{b_1 \to b_2}$ we have

$<c, d> \in R^{a_1 \to a_2,\ b_1 \to b_2}$ iff $\forall c', d'. <c', d'> \in R^{a_1,\ b_1} \supset <cc', dd'> \in R^{a_2,\ b_2}$

(LR.3) For all $<f, g> \in R^{T \Rightarrow T}$ and $<c, d> \in Dom_{\mathcal{U}}^{\forall f} \times Dom_{\mathcal{B}}^{\forall g}$ we have

$<c, d> \in R^{\forall f,\ \forall g}$ iff $\forall a, b. <a, b> \in R^T \supset <ca, db> \in R^{f(a),\ g(b)}$

(LR.4) For functions $<f, g> \in R^{T \Rightarrow T}$ and $<c, d> \in Dom_{\mathcal{U}}^{\exists f} \times Dom_{\mathcal{B}}^{\exists g}$, we have

$<c, d> \in R^{\exists f,\ \exists g}$ iff for all $<a, b> \in R^T$ and $<c_1, d_1> \in R^{\forall t. ft \to a,\ \forall t. gt \to b}$,

$<(Sum_{a,\ f}\ c_1)c,\ (Sum_{b,\ g}\ d_1)d> \in R^{a,\ b}$.

A logical relation $\mathcal{R}$ over models $\mathcal{U}$ and $\mathcal{B}$ for $\mathcal{SA}(\mathcal{C}_{cst}, \mathcal{C}_{term})$ *preserves* a constant $c \in \mathcal{C}_{term}$ if

$<\mathcal{S}_{\mathcal{U}}(c), \mathcal{S}_{\mathcal{U}}(c)> \in R^{a,\ b}$, where $a = \mathcal{U}[\![\sigma]\!]$, $b = \mathcal{B}[\![\sigma]\!]$, and σ is the type of c.

Note that by the Fundamental Theorem for $\mathcal{TA}$ Relations, the semantic types $a = \mathcal{U}[\![\sigma]\!]$ and $b = \mathcal{B}[\![\sigma]\!]$ of any constant c^σ must be related. Unless otherwise specified, we will assume that any relation $\mathcal{R}$ over models $\mathcal{U}$ and $\mathcal{B}$ for $\mathcal{SA}(\mathcal{C}_{cst}, \mathcal{C}_{term})$ preserves both $\mathcal{C}_{cst}$ and $\mathcal{C}_{term}$.

Let $\mathcal{R}$ be a logical relation over $\mathcal{SA}(\mathcal{C}_{cst}, \mathcal{C}_{term})$ models $\mathcal{U}$ and $\mathcal{B}$. If $\eta_{\mathcal{U}}$ and $\eta_{\mathcal{B}}$ are environments for $\mathcal{U}$ and $\mathcal{B}$, respectively, we say that $\eta_{\mathcal{U}}$ and $\eta_{\mathcal{B}}$ are *related environments with respect to type assignment* A if

$$\eta_{\mathcal{U}}, \eta_{\mathcal{B}} \models A$$

and for every variable x we have

$$<\eta_{\mathcal{U}}(x), \eta_{\mathcal{B}}(x)> \in R^{a,\ b},$$

where $a = \mathcal{U}[\![A(x)]\!]\eta_{\mathcal{U}}$ and $b = \mathcal{B}[\![A(x)]\!]\eta_{\mathcal{B}}$. The "Fundamental Theorem" in [Mitchell and Meyer 85] may be extended to abstract data type declarations.

THEOREM 6. *(Fundamental Theorem of $\mathcal{SA}$ Logical Relations) Let $\mathcal{R}$ be a logical relation over $\mathcal{SA}(\mathcal{C}_{cst}, \mathcal{C}_{term})$ models $\mathcal{U}$ and $\mathcal{B}$ and let $\eta_{\mathcal{U}}$, $\eta_{\mathcal{B}}$ be related environments with respect to type assignment A. Assume that $\mathcal{R}$ preserves the constants of $\mathcal{SA}(\mathcal{C}_{cst}, \mathcal{C}_{term})$. Then for any term M of $\mathcal{SA}_A(\mathcal{C}_{cst}, \mathcal{C}_{term})$,*

$$<\mathcal{U}[\![M]\!]\eta_{\mathcal{U}}, \mathcal{B}[\![M]\!]\eta_{\mathcal{B}}> \in R^{a,\ b},$$

where $a = \mathcal{U}[\![Type_A(M)]\!]\eta_{\mathcal{U}}$ and $b = \mathcal{B}[\![Type_A(M)]\!]\eta_{\mathcal{B}}$

Note that no specific assumptions about **Sum** and **Inj** are made other than those implicit in (LR.4). Furthermore, since our definition (LR.4) is not simply the relation induced by relating constants for **Sum** and **Inj** (in accordance with the treatment of $\exists$-types given in [Mitchell 84c, Bruce, Meyer and Mitchell 85]), the theorem above does not follow from the corresponding theorem in [Mitchell and Meyer 85].

6. Representation Independence for Abstract Data Types

6.1. Implications of the "Fundamental Theorem"

Theorem 6 shows that the meanings of terms are, to a reasonable degree, independent of the representations of the built-in types. That is, if models $\mathcal{U}$ and $\mathcal{B}$ are related by a logical relation, then the meaning of a term M in $\mathcal{U}$ is related to the meaning of the same term M in $\mathcal{B}$. Consequently, if we distinguish programs from other closed terms by choosing some set $\sigma_1, \ldots, \sigma_n$ of closed "program types," then whenever there is a logical relation $\mathcal{R}$ over models $\mathcal{U}$ and $\mathcal{B}$ with

$$Dom_{\mathfrak{A}}^{\sigma_i} = Dom_{\mathfrak{B}}^{\sigma_i}$$

and R^{σ_i} the identity relation, the meaning of any program M in $\mathfrak{A}$ will be identical to the meaning of M in $\mathfrak{B}$.

Since the Fundamental Theorem applies to open terms as well as closed terms, we can also use the theorem to show a degree of representation independence for user-defined data types. More specifically, using the Fundamental Theorem and the substitution lemma, we can show that if P and Q are related data type implementations, then any program using implementation P will give the same result as the same program using Q. Suppose P and Q are closed terms of type $\exists t.\tau$ and M is a term of some program type $\sigma_1, \ldots, \sigma_n$ as above with free variable x of type $\exists t.\tau$. Then whenever we have a relation $\mathfrak{R}$ over models $\mathfrak{A}$ and $\mathfrak{B}$ with each R^{σ_i} the identity relation, and a pair of environments η_a, η_b with

$$\eta_a(x) = \mathfrak{A}[\![P]\!] \quad \text{related to} \quad \eta_b(x) = \mathfrak{B}[\![Q]\!],$$

we have

$$\mathfrak{A}[\![M]\!]\eta_a = \mathfrak{B}[\![M]\!]\eta_b$$

by the Fundamental Theorem. By the Substitution Lemma (see [Bruce and Meyer 84, Bruce, Meyer and Mitchell 85, Mitchell 84c]), we have $\mathfrak{A}[\![M]\!]\eta_a = \mathfrak{A}[\![[P/x]M]\!]$ and similarly for [Q/x]M, so that

$$\mathfrak{A}[\![[P/x]M]\!] = \mathfrak{B}[\![[Q/x]M]\!].$$

Thus logically related data type implementations are observationally equivalent. (With appropriate consideration of environments, this argument also applies to lists $P_1, \ldots, P_n$ and $Q_1, \ldots, Q_n$ of possibly open data type implementations.)

However, the Fundamental Theorem does not give us any information about when the implementations P and Q are related. In particular, if

$$P ::= \textbf{rep}\ \sigma_p\ P_1 \quad \text{and} \quad Q ::= \textbf{rep}\ \sigma_q\ Q_1,$$

then we expect R(P, Q) to depend only on the relationship between the behavior of operation P_1 on σ_p and the behavior of operation Q_1 on σ_q. We will prove this in Section 6.3 using the model construction outlined in Section 6.2.

6.2. Extensions to Models

Let $\mathfrak{A}$ be a second-order model with type $a_0 \in Kind^T$. We define a model $\mathfrak{A}^\dagger a_0$ analogous to the first-order $\mathfrak{A}^+$ of Section 2.3. Intuitively, we want $\mathfrak{A}^\dagger a_0$ to be $\mathfrak{A}$ with an extra copy of a_0, so that we can distinguish between the type a_0 when it is used as the representation type for some abstract type, and the type a_0 when it is used otherwise.

Essentially, we let $\mathcal{K}ind^\dagger$ be the kind structure generated by elements of $\mathcal{K}ind$, plus a new type a_1. One way to describe this construction is begin with elements of $\mathcal{K}ind$ as constants, plus a new type constant a_1, and take equivalence classes of closed constructor expressions as elements of $\mathcal{K}ind^\dagger$. We consider $\mu = \nu$ if this equation is provable from the equational theory of $\mathcal{K}ind$, using the usual inference rules and the new rule

$$\frac{\mu a_0 = \nu a_0}{\mu a_1 = \nu a_1}$$

This gives us a kind structure in which a_1 satisfies precisely the same equations as a_0, but we do not have $a_0 = a_1$. Furthermore, any element of $Kind^{\dagger T}$ is of the form fa_1, where f is a (possibly constant) function from $Kind^T$.

We define $\mathcal{D}om^\dagger$ using only the sets of $\mathcal{D}om$, with Dom^{a_0} used for $Dom^{\dagger a_1}$. More generally, for any type fa_1, we let $Dom^{\dagger fa_1}$ be Dom^{fa_0}. It is not hard to verify that this construction gives us a second-order model.

6.3. A Characterization of Observational Equivalence

We can show that the meanings

$$\text{Inj } a_0\ c = \mathfrak{A}[\![\ \text{rep } \sigma\ M]\!]\eta_{\mathfrak{A}} \quad \text{and} \quad \text{Inj } b_0\ d = \mathfrak{B}[\![\ \text{rep } \tau\ N]\!]\eta_{\mathfrak{B}}$$

of two implementation expressions are logically related iff there is a correspondence between the behavior of c on a_0 and the behavior of d on b_0. This characterization of a logical relation $\mathcal{R}$ on existential types uses the extended models $\mathfrak{A}^\dagger a_0$ and $\mathfrak{B}^\dagger b_0$ with "duplicate copies" of a_0 and b_0. From this characterization, it is easy to describe when the meanings of expressions of the form $\lambda\vec{s}\ \lambda\vec{x}.$ rep σ M will be logically related.

THEOREM 7. *Let $\mathcal{R}$ be a logical relation over $\mathfrak{A}$ and $\mathfrak{B}$, with $a_0 \in Kind_{\mathfrak{A}}{}^T$ and $b_0 \in Kind_{\mathfrak{B}}{}^T$. Let $<f, g> \in R^{T\Rightarrow T}$, so that $\exists f$ and $\exists g$ are related types, and let c, d be elements of $\mathfrak{A}$ and $\mathfrak{B}$ with $c \in Dom_{\mathfrak{A}}{}^{a_0}$ and $d \in Dom_{\mathfrak{B}}{}^{g b_0}$. (Note that a_0 and b_0 need not be related by $\mathcal{R}$.) Then $<$Inj a_0 c, Inj b_0 d$> \in R^{\exists f, \exists g}$ iff there is a logical relation S over extended models $\mathfrak{A}^\dagger a_0$ and $\mathfrak{B}^\dagger b_0$ such that $<c, d> \in S^{f a_1,\ g b_1}$ and $S^{a, b} = R^{a, b}$ for every $<a, b> \in R^1$.*

7. Conclusion and Directions for Further Investigation

Theorems 6 and 7 demonstrate some important representation independence properties. Informally, Theorem 6 says that logically related data type implementations cannot be distinguished by programs, while Theorem 7 assures us that whenever there is a correspondence between the operations of P and Q, implementations P and Q will be logically related. Since it follows from these theorems that implementations based on different representation types will be indistinguishable as long as the operations correspond properly, it seems fair to conclude that SOL is type secure. (In fact, Theorems 6 and 7 might be taken as a formal definition of type security.) One hopes that other languages such as Pebble and Standard ML can be proved type secure in an equally satisfying precise sense.

It would be useful to be able to prove that data type implementations we encounter in practice are observationally equivalent. In principle, this may be done using the inference system in [Mitchell and Meyer 85] for demonstrating that pairs of terms are logically related. However, we have not tried to carry out any significant proofs. One interesting problem here is that the P and Q may be observationally equivalent if we decide to keep all other declarations fixed, but P and Q may become observably different when some other declarations are changed. Thus, the interdependence of data types becomes important. A useful proof system for demonstrating observational equivalence of data type implementations seems an important area for future research.

Acknowledgements: Thanks to Albert Meyer and Oliver Schoett for some helpful discussions. The investigative framework of this paper is based on joint work with Meyer.

References

[Bruce and Meyer 84] Bruce, K. and Meyer, A., A Completeness Theorem for Second-Order Polymorphic Lambda Calculus. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)* , 1984, pages 131-144..

[Bruce, Meyer and Mitchell 85] Bruce, K.B., Meyer, A.R. and Mitchell, J.C., The semantics of second-order lambda calculus. to appear

[Burstall and Lampson 84] Burstall, R. and Lampson, B., A Kernel Language for Abstract Data Types and Modules. In *Proc. Int'l Symp. on Semantics of Data Types* , 1984, pages 1-50.

[Donahue 79] Donahue, J., On the semantics of data type. *SIAM J. Computing* 8 1979. pages 546-560

[Friedman 75] Friedman, H., Equality Between Functionals. In R. Parikh (ed.), *Logic Colloquium*, pages 22-37. Springer-Verlag 1975.

[Girard 71] Girard, J.-Y., Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In Fenstad, J.E. (ed.), 2^{nd}

Scandinavian Logic Symp., pages 63-92. North-Holland 1971.

[Gordon, et. al. 79] Gordon, M.J., R. Milner and C.P. Wadsworth, *Edinburgh LCF*. Lecture Notes in Computer Science, Vol. 78 Springer-Verlag 1979.

[Haynes 84] Haynes, C.T., A Theory of Data Type Representation Independence. In *Int. Symp. on Semantics of Data Types* , Springer-Verlag , 1984, pages 157-176.

[Liskov et. al. 81] Liskov, B. et. al., *CLU Reference Manual*. Lecture Notes in Computer Science, Vol. 114 Springer-Verlag 1981.

[MacQueen 85] MacQueen, D.B., Modules for Standard ML. *Polymorphism* 2, 2 1985. 35 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[MacQueen 86] MacQueen, D.B., Using dependent types to express modular structure. In *Proc. 13-th ACM Symp. on Principles of Programming Languages* , 1986. To appear.

[Martin-Löf 79] Martin-Löf, P., Constructive mathematics and computer programming. 1979. Paper presented at 6th International Congress for Logic, Methodology and Philosophy of Science, Preprint, Univ. of Stockholm, Dept. of Math. 1979

[McCracken 79] McCracken, N., *An Investigation of a Programming Language with a Polymorphic Type Structure*. Syracuse Univ. 1979.

[Milner 85] Milner, R., The standard ML core language. *Polymorphism* 2, 2 1985. 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[Mitchell 84c] Mitchell, J.C., Semantic models for second-order lambda calculus. In *Proc. 25-th IEEE Symp. on Foundations of Computer Science* , 1984, pages 289-299.

[Mitchell and Plotkin 85] Mitchell, J.C. and Plotkin, G.D., Abstract types have existential types. In *Proc. 12-th ACM Symp. on Principles of Programming Languages* , January, 1985. pp. 37-51.

[Mitchell and Meyer 85] Mitchell, J.C. and Meyer, A.R., Second-order logical relations. In *Logics of Programs* , June, 1985. pages 225-236.

[Plotkin 80] Plotkin, G.D., Lambda definability in the full type hierarchy. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363-373. Academic Press 1980.

[Reynolds 74] Reynolds, J.C., Towards a Theory of Type Structure. In *Paris Colloq. on Programming* , Springer-Verlag , 1974, pages 408-425.

[Reynolds 83] Reynolds, J.C., Types, Abstraction, and Parametric Polymorphism. In *IFIP Congress* , 1983.

[Statman 82] Statman, R., Logical relations and the typed lambda calculus. (Manuscript.) To appear in Information and Control.

[Tait 67] Tait, W.W., Intensional interpretation of functionals of finite type. *J. Symbolic Logic* 32 1967. pages 198-212