# On Extending Computational Adequacy by Data Abstraction*

*Val Breazu-Tannen* *Ramesh Subrahmanyam*
Department of Computer and Information Science
University of Pennsylvania
200 South 33rd St., Philadelphia, PA 19104, USA

E-mail: val@cis.upenn.edu, ramesh@saul.cis.upenn.edu

**Abstract:** Given an abstract data type (ADT), an algebra that *specifies* it, and an implementation of the data type in a certain language, if the implementation is "correct", a certain principle of *modularity of reasoning* holds. Namely, one can safely reason about programs in the language extended by the ADT, by interpreting the ADT operation symbols according to the specification algebra. The main point of this paper is to formalize correctness as a local condition involving only the specification and the implementation and to prove the equivalence of such a condition to the modularity principle. We conduct our study in the context of a language without divergence ( in subsection 2.1), and for languages with divergence and general recursion (in subsections 2.2 and 2.3). We also describe a sufficient condition under which, given an implementation, there may be a finite set of observational equivalences which imply the local condition. Further, we illustrate a technique for proving in a practical situation that a given implementation of an abstract data type is correct.

## 1 Introduction

This work belongs to a line of investigation initiated by Reynolds [Rey74, Rey83] and is a natural continuation of that of Mitchell and Meyer [MM85, Mit86]. (See also [Don79, Hay84].) Their results support, in Reynolds' words, "the thesis that type structure is a syntactic discipline for maintaining levels of abstraction". Reynolds [Rey83] shows that programs interpreted using different semantic representations for built-in datatypes have the same meaning provided the representations are suitably related. Mitchell [Mit86] shows that programs using different implementations for user-defined datatypes have the same meaning provided the semantics of the representations are suitably related. "Suitably related" refers to *logical relations* [Plo80, Fri75, Sta85]. We interpret these results as saying that the levels of abstraction are properly maintained in the semantics, in other words, these are *semantic modularity* results.

The main point of our paper is that *modularity of reasoning* also holds. The implementation of a user-defined abstract data type is written as the programmer has in mind an intended interpretation (a *specification*) for the data items and the operations on them. If the implementation is *correct* with respect to the specification, then the programmer expects that when reasoning about programs containing ADT operations one can ignore the implementation and use only the specification. But what should we take as the definition of "correct"?

For example, consider a datatype **Sets-of-nats**,

*The authors are partially supported by ONR Grant NOOO14-88-K-0634, and by NSF Grant CCR-90-57570.

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0161...$1.50

with operations including **emp**, **ins**, **del** and **mem**, among others. The specification we have in mind is the algebra whose elements are the finite sets of natural numbers and in which the operations are the usual empty-set, insertion, deletion, and membership test. Consider for example an ADT implementation of this datatype in which the sort **Set** is implemented by the type $(\mathbf{nat} \to \mathbf{bool}) \times \mathbf{List}$. The idea is to "store" an element in the function part if the element is less than 100, and to store it in the list part otherwise. The implementations of the operation symbols are:

- $\mathbf{emp}^* = (\lambda x.\ \mathbf{false}, \mathbf{nil})$
- $\mathbf{ins}^*$, given $n$ and $S(\equiv (S_f, S_l))$ does the following. If $n \geq 100$ it returns $(S_f, \mathbf{cons}(n, S_l)$; otherwise it returns $(S_f^+, S_l)$, where $S_f^+$ has the same value at all points except at $n$ where its value is **true.**
- $\mathbf{del}^*$ is defined as follows: given arguments $n$ and $S$, where $S = (S_f, S_l)$, if $n \geq 100$, it returns $(S_f, S_l^-)$, where $S_l^-$ is obtained by deleting all occurrences of $n$ from $S_l$. If $n < 100$ it returns $S_f^-, S_l)$, where $S_f^-$ has the the same value as $S_f$ at all points except $n$ where its value is **false.**
- $\mathbf{mem}^*$, on arguments $n$ and $S(\equiv (S_f, S_l))$ checks to see if $n \geq 100$; if it is then it returns true if $n$ is in $S_l$, and false otherwise. If $n < 100$ it returns $S_f(n)$.

According to the specification, the insertion operation is idempotent

$$(1) \qquad \mathbf{ins}(x, \mathbf{ins}(x, s)) = \mathbf{ins}(x, s)$$

while in the implementation, the **cons** part is not idempotent. What then, is the basis for the programmer's intuition that programs that differ according to the equation (1) evaluate to the same result? (That is, (1) is an *observational equivalence.*) Clearly, well-typing is necessary, otherwise we can put expressions of ADT type in contexts that process lists and therefore we can operationally detect the failure of idempotence. But this is not enough. Consider an implementation of $\mathbf{del}(x, s)$ as the function that on the list part eliminates only the first occurrence of $x$ it finds. Clearly this implementation should not be considered correct, and in fact we can see that the equation (1) fails to be an observational equivalence as $\mathbf{mem(100, del(100, ins(100, ins(100, emp))))}$ and $\mathbf{mem(100, del(100, ins(100, emp)))}$ evaluate to different observable results. We will show in Section 3 that the implementation described above is correct.

Let $\mathcal{L}$ be a base language in which we implement an ADT given by a many-sorted algebraic signature $\Sigma$. In this paper we will stay away from the issue of choosing between ADT specification paradigms (initial, final, etc.) and take specifications to be algebras. Let $\mathcal{A}$ be a specification algebra for the ADT, that is, a $\Sigma$-algebra. To give meaning to programs "according to the specification" we take models of $\mathcal{L} + \Sigma$ in which $\mathcal{A}$ is fully and faithfully embedded[1]. This is justified by the fact that the $\mathcal{L}+\Sigma$-equational theory of such models is a conservative extension of the $\Sigma$-equational theory of $\mathcal{A}$. Now, whatever definition we take for correctness of implementations wrt specifications, we want the following to hold:

**Modularity of Reasoning Principle.** *Any* $(\mathcal{L} + \Sigma)$*-model in which the specification algebra is fully and faithfully embedded is sound for reasoning about* $(\mathcal{L} + \Sigma)$*-observational equivalence.*

This principle relates two important concepts in programming languages: that of *data abstraction* and that of *computational adequacy.* Computational adequacy (see for example [MC88]) is a criterion of good fit between denotational and operational semantics, and it allows reasoning about observational equivalence by checking denotational equality. The principle above insists that with data abstraction there be an adequate denotational semantics in which the abstract data type is interpreted as a prescribed specification. This cannot happen if the base language $\mathcal{L}$ does not have an adequate denotational semantics as well. Thus, we should be able to *extend* the adequate semantics of $\mathcal{L}$ using the specification and get an adequate semantics for $\mathcal{L} + \Sigma$.

[1]See subsection 2.1.

All this will not be possible, of course, if the implementation is arbitrary. Going back to the definition of correctness of implementations wrt specifications, it is natural to ask: why not take the modularity of reasoning principle itself as the definition of correctness? The philosophical objection to this, in view of the discussion above on extension of adequacy, is that it appears to be too *global* a condition since it refers to the whole language $\mathcal{L}+\Sigma$. This objection is overcome by the first version of our main result, Theorem 1, which establishes that this global principle is equivalent to a *local* condition that relates only the specification algebra and the operational behavior of the implementation of the ADT (subsection 2.1). The condition states that any equation between closed $\Sigma$-terms that holds at observable sorts in the specification algebra must hold operationally in the implementation. Therefore, we feel justified in taking the modularity of reasoning principle (or the equivalent local condition) as the definition of correctness of implementations wrt specifications.

This local condition is the basis for a verification technique that we illustrate on an example in section 3. The local condition, however, requires us to verify an infinite set of closed equations. We describe, therefore, a sufficient condition under which, given an implementation, there is a finite set of observational equivalences, and verifying these guarantees that the local condition is satisfied.

The language we consider for the first version of our main result (Theorem 1) is quite simple, namely a simply typed lambda calculus with products and first-order constants. We prove however (Theorem 2 in subsection 2.2) that essentially the same result holds if we consider a simply typed lambda calculus with recursion and with a call-by-name evaluator, and we continue to observe values of base type (this is a language similar to Plotkin's PCF [Plo77]). Moreover, we sketch how to prove yet another version of our main result, this time for a call-by-value evaluator, and observing termination at all types (subsection 2.3). This attests to the robustness of the concepts examined.

## 2 Data Abstraction and Modularity of Reasoning

### 2.1 Data Abstraction in a Language without Recursion

**Observables.** We fix a many-sorted signature $\mathcal{O}$ whose sorts and operation symbols we call "observable". We also fix an $\mathcal{O}$-algebra as a *standard interpretation* for the observables. and we assume that $\mathcal{O}$ contains distinct ground constants denoting each element of the standard interpretation (so called *numerals*).

**Base language.** We fix a language $\mathcal{L}$ which is a simply typed lambda calculus with products and which extends $\mathcal{O}$, that is, the observable sorts are the base types and the observable operations are the constants of various types (hence we have only first-order constants).

By an *$\mathcal{L}$-model* we mean a type-frame model in the sense of Friedman [Fri75] which is moreover such that the interpretation of the base types (observable sorts) and that of the constants (observable operation symbols) is the same as the standard interpretation of $\mathcal{O}$.

Moreover, we fix an *evaluator* (a total function on closed terms) for $\mathcal{L}$, $Eval_{\mathcal{L}}$, assuming only that there exists an $\mathcal{L}$-model in which the evaluator is sound ($[\![Eval_{\mathcal{L}}(M)]\!] = [\![M]\!]$ for any closed term, $M$, of observable type).

**ADT extension.** We fix an *ADT-signature*, that is, a many-sorted algebraic signature $\Sigma$ and, for simplicity, we assume that $\Sigma$ contains $\mathcal{O}$ [2].

We fix a *specification algebra* $\mathcal{A}$, that is, a $\Sigma$-algebra about which we only require that the interpretation of the observable sorts and that of the observable operation symbols is also the same as the standard interpretation of $\mathcal{O}$.

We will be interested in the *extended* language $\mathcal{L}+\Sigma$, which is also a simply typed lambda calculus with products and first-order constants. $\mathcal{L}+\Sigma$ extends $\mathcal{O}$, since it extends $\Sigma$ and $\Sigma$ contains $\mathcal{O}$.

[2]At the expense of additional complexity in the presentation, we could have $\Sigma$ contain only part of the observables, as well as give $\mathcal{L}$ more base types and first-order constants than the observables. We could also ask that the numerals be just closed terms of the observable signature rather than distinct ground constants, and thus avoid infinite signatures.

$\mathcal{L} + \Sigma$-models are Friedman models in which the interpretation of the observables is again the same as the standard one.

We fix an *implementation* of $\Sigma$ in the base language $\mathcal{L}$, that is, a mapping that associates
-to every sort $s$ of $\Sigma$ an *implementation type* $s^*$ of $\mathcal{L}$ (not necessarily a base type) such that for any observable sort $o^* = o$, and
-to every operation symbol $f : s_1 \times \cdots \times s_k \to s$ in $\Sigma$ an *implementation term* which is a closed term of $\mathcal{L}$, $f^* : s_1^* \times \cdots \times s_k^* \to s^*$, such that for any observable operation symbol $q^* = q$.

The implementation, together with the evaluator of $\mathcal{L}$, gives an evaluator for $\mathcal{L} + \Sigma$, namely, we extend the implementation translation to a translation $\mathcal{L} + \Sigma \to \mathcal{L}$, $\tau \mapsto \tau^*$ on types and $M \mapsto M^*$ on terms and then put $Eval_{\mathcal{L}+\Sigma}(M) \overset{\text{def}}{=} Eval_{\mathcal{L}}(M^*)$. Note that in general $M : \tau$ while $Eval_{\mathcal{L}+\Sigma}(M) : \tau^*$. However, if $\tau$ is observable then $M$ and $Eval_{\mathcal{L}+\Sigma}(M)$ have the same type.

**Theorem 1** *The following are equivalent:*

**(a : Modularity of Reasoning)** *Any $(\mathcal{L} + \Sigma)$-model $\mathcal{M}$ in which the specification algebra $\mathcal{A}$ is fully and faithfully embedded, is sound for reasoning about $(\mathcal{L} + \Sigma)$-observational equivalence, that is, for any $(\mathcal{L} + \Sigma)$-terms $M, N$*

$$\mathcal{M} \models M = N \quad \Rightarrow \quad M \cong_{\mathcal{L}+\Sigma} N$$

**(b : Local Condition for Correctness)** *For any closed algebraic $\Sigma$-terms of observable sort $t_1, t_2$, if $\mathcal{A} \models t_1 = t_2$ then $t_1^* \cong_{\mathcal{L}} t_2^*$ (equivalently, $Eval_{\mathcal{L}}(t_1^*) = Eval_{\mathcal{L}}(t_2^*)$).*

**Remarks.**
1. By full and faithful embedding of a $\Sigma$-algebra $\mathcal{A}$ in an $\mathcal{L} + \Sigma$-model $\mathcal{M}$ we mean that in $\mathcal{M}$ the base types are interpreted as the corresponding carriers of $\mathcal{A}$ and the first-order constants are interpreted as the corresponding operations in $\mathcal{A}$. Crucial property: an equation between algebraic $\Sigma$-terms holds in $\mathcal{A}$ iff it holds in $\mathcal{M}$.
2. The notion of observational equivalence $\cong_{\mathcal{L}}$ is the usual one, namely that two terms are observationally equivalent if whenever put in the same program (closed term of observable type) context, the resulting programs evaluate to the same observable result.

**Proof of Theorem 1.** Clearly **(a)** implies **(b)**. To show that **(b)** implies **(c)**, we proceed along by the following steps:

1. Given an implementation $\mathcal{I}$ and a model $\mathcal{M}$ of the $\mathcal{L} + \Sigma$, we define an $\mathcal{L} + \Sigma$-model $\mathcal{M}'$ as follows:

   (a) Consider the translation $()^*$ of $\mathcal{L} + \Sigma$-types into $\mathcal{L}$-types mentioned before. Given an $\mathcal{L} + \Sigma$-type $\tau$, it substitutes every occurence of a sort symbol $s$ in $\tau$ by the implementation of that sort using $\mathcal{I}$. The interpretation of type $\tau$ in $\mathcal{M}'$, $D'_\tau$ is the set that is the interpretation of $\tau^*$ in $\mathcal{M}$, that is $D_{\tau^*}$.

   (b) The application relation $\cdot_{\mathcal{M}'}$ for $\mathcal{M}'$ is defined as follows: for $d \in D'_{\sigma\to\tau}$ and $e \in D'_\sigma$, $d \cdot_{\mathcal{M}'} e = d \cdot_{\mathcal{M}} e$. This makes sense because, by construction, $d \in D_{\sigma\to\tau}$ and $e \in D_\sigma$.

   (c) The meaning function for $\mathcal{L} + \Sigma$-terms in $\mathcal{M}'$ when given $\mathcal{L} + \Sigma$-term $M$ returns $[\![M^*]\!]_{\mathcal{M}}$, where $M^*$ is obtained by replacing every $\Sigma$-symbol in $M$ by its translation given by $\mathcal{I}$ (the translation of $\mathcal{L} + \Sigma$-terms mentioned earlier).

   It is routine to verify that the above defines a $\mathcal{L} + \Sigma$-model.

2. The next thing to prove is that if $M$ is a $\mathcal{L}+\Sigma$-term of type $\tau$ then $[\![M]\!]_{\mathcal{M}'}, [\![M^*]\!]_{\mathcal{M}} \in D'_\tau = D_{\tau^*}$, and furthermore $[\![M]\!]_{\mathcal{M}'} = [\![M^*]\!]_{\mathcal{M}}$. This is proved by induction on the structure of $M$.

3. We can define a logical relation $\mathbf{R}$ between $\mathcal{M}$ and $\mathcal{M}'$ by defining the relation on the base types. On observable base types the relation is identity. On a base type that is the interpretation of sort symbol $s$, define the relation as follows:

   $d \in D_s \mathbf{R} d' \in D'_s \Leftrightarrow \exists t \cdot \; d = [\![t]\!]_{\mathcal{M}} \wedge d' = [\![t]\!]_{\mathcal{M}'}$

   where $t$ above is a closed $\Sigma$-term.

4. If condition (b) of the theorem holds then $\mathbf{R}$ relates the meanings of the constants in $\Sigma$ in the the two models. To show this we have to show that for every $f \in \Sigma$, $\forall d_1,..,d_n,d'_1,..,d'_n.\ d_1\mathbf{R}d'_1,..,d_n\mathbf{R}d'_n \Rightarrow$
$$[\![f]\!]_{\mathcal{M}}(d_1,..,d_n)\mathbf{R}[\![f]\!]_{\mathcal{M}'}(d'_1,..,d'_n)$$
By definition, for each $i$ there is a closed $\Sigma$-term $t_i$ such that $[\![t_i]\!]_{\mathcal{M}} = d_i$ and $[\![t_i]\!]_{\mathcal{M}'} = d'_i$. Therefore, $[\![f]\!]_{\mathcal{M}}(d_1,..,d_n) = [\![f(t_1,..,t_n)]\!]_{\mathcal{M}}$ and $[\![f]\!]_{\mathcal{M}'}(d'_1,..,d'_n) = [\![f(t_1,..,t_n)]\!]_{\mathcal{M}'}$. So $[\![f]\!]_{\mathcal{M}}(d_1,..,d_n)\mathbf{R}[\![f]\!]_{\mathcal{M}'}(d'_1,..,d'_n)$.

   If the result sort of $f$ is an observable sort, say **nat**, and $[\![f]\!]_{\mathcal{M}}(d_1,..,d_n) = n$, then by definition $\mathcal{A} \models f(t_1,..,t_n) = \mathbf{n}$. Using condition (b), $(f(t_1,..,t_n))^* \cong_{\mathcal{L}} \mathbf{n}$. Using the adequacy of $\mathcal{M}$ for $\mathcal{L}$, and hence the adequacy of $\mathcal{M}'$, it follows that $\mathcal{M}' \models (f(t_1,..,t_n))^* = \mathbf{n}$. Therefore $[\![f]\!]_{\mathcal{M}'}(d'_1,..,d'_n) = n$.

5. Given condition (b) $\mathbf{R}$ is a logical relation that relates constants in $\Sigma$. By the fundamental theorem of logical relations [Sta85], for any $\mathcal{L}+\Sigma$-term, its meanings in the two models are related.

   To complete the proof of the theorem, suppose $\mathcal{M} \models M = N$. Given any $\mathcal{L}+\Sigma$ context $C[]$ of observable type, $\mathcal{M} \models C[M] = C[N]$. Since $\mathbf{R}$ is an identity at observable types, and relates the meanings of terms in the two models, $\mathcal{M}' \models C[M] = C[N]$. So $\mathcal{M} \models (C[M])^* = (C[N])^*$, and so $Eval_{\mathcal{L}+\Sigma}(C[M]) = Eval_{\mathcal{L}+\Sigma}(C[N])$. Since this is the case for an arbitrary context $C[]$, $M \cong_{\mathcal{L}+\Sigma} N$.

▮

## 2.2 Call-by-Name Language with Recursion

In this subsection we consider implementations in a language with general recursion. This requires us to redefine specification algebras to be continuous algebras, that is to have cpo's for their carriers, so that we can give meanings to general recursive computations over the specification algebra. In such algebras the strictness information of the various operations is also present. We find it convenient to explicitly talk about the "divergent element" in the algebra, and so we require the signature of the algebra to include a constant $\perp_s$ for each sort $s$. Another major difference is that we will want to work with models in which not only is the evaluator sound, but which are also *adequate* that is, divergent terms mean bottom.

We will describe the language, the observables and the abstract data types in this subsection by mentioning merely those aspects that are new or different from those in subsection 2.1.

**Observables** For each sort $s$ we have a distinguished nullary constant symbol of that sort, $\perp_s$. The standard interpretation for the algebra of observables has as its carriers flat CPO's, and the interpretation of the operation symbols are continuous. The constant $\perp_s$ denotes the bottom element of the carrier of sort $s$.

**Base Language** The language also has the fixpoint recursion operator $Y$. We only consider one model $\mathcal{C}$ for the language, namely, the full continuous type hierarchy over observable types, with product interpreted as the (non-smash) product. We fix a call-by-name evaluator for this language. The evaluator does not evaluate under the pairing constructor. We do not specify how the evaluator evaluates the rest of the language; we require that soundness and adequacy at base types holds with respect to $\mathcal{C}$.

**ADT Extension** We fix an ADT signature, ensuring that there is a constant $\perp_s : s$ for every sort $s$. We will call it $\Sigma_\perp$ to remind the presence of $\perp_s$ in the signature.

We are now interested in the extended language $\mathcal{L}Y+\Sigma_\perp$. Its models have as observable types the carriers of the standard observable algebra, and cpo's as interpretations for the sorts in $\Sigma_\perp$. The rest of the model consists of the continuous type hierarchy over these cpo's.

The notion of an implementation is identical to that described in subsection 2.1, barring the fact that the implementation of $\perp : s$ has to be $Y(\lambda x{:}s.x)$.

A specification algebra is now a continuous $\Sigma_\perp$-algebra $\mathcal{A}$ (given signature $\Sigma_\perp$),(i.e.) it has cpo's for its carriers, interprets the operator symbols in $\Sigma_\perp$ as continuous functions, and $\perp_s$ as the bottom element of the carrier of $s$.

Let $\mathcal{D}$ and $\mathcal{E}$ be cpo's and $\mathbf{R} \subseteq \mathcal{D} \times \mathcal{E}$ be a relation. Due to the presence of $\perp$ in the signature, every $\Sigma_\perp$-homomorphic relation will be *strict*, that is, $\perp_\mathcal{D} \mathbf{R} \perp_\mathcal{E}$. $\mathbf{R}$ is said to be *continuous* if and only if for any two directed subsets $D$ and $E$ of $\mathcal{D}$ and $\mathcal{E}$ respectively, if $\forall d \in D. \exists e \in E.\ d \mathbf{R} e$ and $\forall e \in E. \exists d \in D.\ d \mathbf{R} e$, then $(\sqcup_\mathcal{D} D) \mathbf{R} (\sqcup_\mathcal{E} E)$ [3].

**Theorem 2** *The following are equivalent:*

*(a) Any ($\mathcal{L}Y+\Sigma_\perp$)-model $\mathcal{M}$ in which the specification algebra $\mathcal{A}$ is fully and faithfully embedded is sound for reasoning about ($\mathcal{L}Y + \Sigma_\perp$)-observational equivalence, that is, for any ($\mathcal{L}Y + \Sigma_\perp$)-terms $M, N$*

$$\mathcal{M} \models M = N \;\Rightarrow\; M \cong_{\mathcal{L}Y+\Sigma_\perp} N$$

*(b) For any closed algebraic $\Sigma_\perp$-terms $t_1, t_2$, of observable sort if $\mathcal{A} \models t_1 = t_2$ then $t_1^* \cong_{\mathcal{L}Y} t_2^*$.*

## 2.3 Call-by-Value Language with Recursion

We have considered implementations in a call-by-name language $\mathcal{L}Y$. Let us consider a call-by-value language (i.e) a language with syntax identical to $\mathcal{L}Y$, and whose evaluator differs from the evaluator for $\mathcal{L}Y$ in the fact that application is evaluated in a call-by-value style. The analysis of data abstraction in this setting is very similar to that for $\mathcal{L}Y$, excepting in a few points. We will not present the theorem corresponding to Theorem 2, but make a few technical points.

- The model that we consider is the standard call-by-value model with lifted function types. This model is actually computationally adequate for the evaluator at all types. In this setting, therefore, we are looking to extend this stronger notion of adequacy.
- As in the case of the call-by-name languages, when we define the notion of a model of the strict language extended with additional operations, we need to define the notion of the specification algebra fully and faithfully embedded in the model of the extended language (or even the notion of sameness of the interpretations of the observable sorts and operations in the model and the standard interpretation). This means that the carriers of the various sorts in the algebra are the same partial orders as the interpretation of those sorts in the model.

[3]This is similar to definitions in [Abr90, Ama88].

## 3 Verifying Correctness

### 3.1 Sufficient Condition for Verifying Finitely Many Equations

Given an algebra and an implementation, condition (b) of Theorem 1 involves checking an infinite set of observational congruences. For certain algebras we can come up with finitely many equations, such that verifying that these equations are observational congruences would be equivalent to verifying condition (b). We state a sufficient condition that will guarantee this property.

Consider given an observable signature $\mathcal{O}$ and a standard interpretation, that is an $\mathcal{O}$-algebra, for the observable signature. Let every element in this algebra be the denotation of some closed term over $\mathcal{O}$. Let $\Sigma$ be an extension of $\mathcal{O}$. A set, E, of equations over the signature $\Sigma$ is said to be *sufficient complete* if and only if given any closed $\Sigma$-term, t, of observable sort, there is a closed term $t'$ over $\mathcal{O}$ satisfying $E \vdash t = t'$.

**Theorem 3** *Let $\mathcal{A}$ be a reachable $\Sigma$-algebra in which the interpretation of the sorts and symbols in $\mathcal{O}$ is as in the standard algebra. Let $E$ be a sufficient complete set of equations, each of which is of observable sort, that are valid in $\mathcal{A}$. Any given implementation satisfies condition(b) provided for every equation $t_1 = t_2 \in E$, $t_1^* \cong_\mathcal{L} t_2^*$.*

**Proof:** Let $t_1 = t_2$ be any equation of observable type that is valid in $\mathcal{A}$. Since $E$ is sufficient complete there exist terms $t_1'$ and $t_2'$ over the observable signature such that $E \vdash t_1 = t_1'$ and $E \vdash t_2 = t_2'$. clearly $\mathcal{A} \models t_1' = t_2'$ and since $t_1'$ and $t_2'$ are terms over the observable signature, $(t_1')^* \cong_\mathcal{L} (t_2')^*$. It follows from the hypothesis of the theorem and by induction on the length of the equational proof of $t_1 = t_1'$ that $(t_1)^* \cong_\mathcal{L} (t_1')^*$.

Likewise $(t_2)^* \cong_{\mathcal{L}} (t_2')^*$. Piecing the chain together:

$$(t_1)^* \cong_{\mathcal{L}} (t_1')^* \cong_{\mathcal{L}} (t_2')^* \cong_{\mathcal{L}} (t_2)^*$$

▮

**Remark:** This theorem also extends to the situation where the ADT is given by a continuous $\Sigma$-algebra and the implementation is done in a language that features general recursion. In this case it is assumed that $\Sigma$ contains a constant $\perp_s$ for each sort $s$ which denotes the bottom element of that sort. Sufficient completeness is the same as before, modulo the fact that $\perp$ may appear in $\mathcal{O}$.
▮

## 3.2 An Example

We will illustrate how one might prove that an implementation of an ADT is proved correct using this sufficient condition. We will consider the ADT of finite sets with the interpretation of **emp**, **ins**, **del**, and **mem** being the empty set, the standard insertion, deletion and membership test operations, respectively. As mentioned before, there will be a constant $\perp_s$ for each sort $s$ which will denote the bottom element of that sort.

The following equations are valid in this algebra:
(1) $\mathbf{mem}(x,\mathbf{emp}) = \mathbf{false}$
(2) $\mathbf{mem}(x,\mathbf{ins}(y,s)) = \mathbf{or}(\mathbf{eq}(x,y),\mathbf{mem}(x,s))$
(3) $\mathbf{mem}(x,\mathbf{del}(y,s)) =$
$\mathbf{and}(\mathbf{not}(\mathbf{eq}(x,y)),\mathbf{mem}(x,s))$

These equations can be shown to be sufficient complete: when oriented left to right they are Church-Rosser and strongly normalizing. Furthermore, any normal form of observable type is easily proved to be a numeral (if the type is **nat**), or **true** or **false** if the type is **bool**. This can be shown by an induction on the structure of terms; however, we need a hypothesis for terms of type **Set**. We can show that every normal-form term of type **Set** is in the language defined by the grammar below:

$C ::= \mathbf{emp} \mid \mathbf{ins}(\mathrm{n}, C) \mid \mathbf{del}(\mathrm{n}, C)$

**Remarks:**

1. In general, in carrying out such a a proof, one isolates a subset of the signature consisting of the constants in the observable signature, and constants from the rest of the signature whose result sort is a non-observable sort: let us call them constructors. One then shows that every normal-form of observable sort is generated by the observable signature, and every normal form of non-observable sort is generated from constructors.

2. There is a sense in which the equations above were "systematically discovered". We first identified the constructors,(**emp**, **ins**,and **del**) and then tried to discover defining equations for the remaining operations (**mem**) which were primitive recursive definitions over the constructors. ▮

If we consider the implementation of finite sets described in the introduction, we can verify that for each equation $t_1 = t_2$ above, the expression $t_1^*$ is observationally equivalent to the expression $t_2^*$. This verification can be done either by hand, or by using a theorem-prover.

We can complicate the situation above by considering as the ADT the algebra of finite sets with a bottom element, with operations **ins**, **del** and **mem** interpreted by the standard set insertion, deletion and membership test operations (which are strict in their arguments). As for the observable algebra, all operations are taken to be the standard ones which are strict in their arguments. The equations that we wrote down for the previous case do not apply anymore: in particular the equation

$$\mathbf{mem}(x,\mathbf{emp}) = \mathbf{false}$$

is not valid in the algebra. Every instance of the above equation where $x$ is replaced by a numeral is valid however. The infinite set of instances of this equation, equations (2), (3) and (4) above, and the following equations can be shown to constitute a sufficient complete set of equations.

- $\mathbf{mem}(\perp, s) = \perp$
- $\mathbf{mem}(x, \perp) = \perp$

In this case it would suffice to verify the equation $\mathbf{mem}^*(\mathbf{0},\mathbf{emp}^*) = \mathbf{false}$, and the conditional equation $\mathbf{mem}^*(x,\mathbf{emp}^*) = \mathbf{false} \Rightarrow \mathbf{mem}^*(\mathrm{s}(x),\mathbf{emp}^*) = \mathbf{false}$.

## 4 Comparison with Related Work

Among all the earlier work we mentioned in the introduction [Rey74, Rey83] [MM85, Mit86] [Don79, Hay84] our paper comes closest in spirit to Mitchell's [Mit86]. However, Mitchell's paper is about relating the denotational semantics of different implementations of the same ADT signature such that the semantics of programs of observable type doesn't change. Our paper is about relating the operational behavior of an implementation to denotations in a specification algebra such that the specification can form a basis for reasoning about observational equivalence in the extended language. The two papers have in common the technique of logical relations, though in our paper logical relations do not seem to play a conceptual role; they are merely a tool.

We present in some detail a technique for verifying correctness of an implementation for a given specification. Mitchell states the desirability of techniques for verifying that two given implementations of an ADT are suitably related (and hence interchangeable). Since the paper uses SOL, a language based on the Girard-Reynolds polymorphic lambda calculus [MP85], the issue comes down to the existence of second-order logical relations [MM85] which is still problematic. We believe that the idea of our verification technique can in fact be adapted to show relatedness of implementations, provided we consider predicative versions of the language.

Most importantly, our goal is an extension of computational adequacy and we consider languages with recursion and languages in which the evaluation order makes a difference. This requires new techniques beyond the standard logical relations.

## 5 Directions for Future Research

It will be useful to carry out this research program in other settings: one candidate is nondeterministic abstract data types. A more challenging setting is languages with state: the paradigm of specifications/implementations as algebras is not sufficient.

Another issue that needs to be studied is that of nested abstract data types. It will be useful to have a facility in the language to ensure the checking of type correctness of programs in the presence of such nested abstract data types. Plotkin and Mitchell's language **SOL** [MP85] is an example of one such language. Carrying out a program of this nature for full Girard-Reynolds polymorphism is an interesting challenge. One could also consider related but predicative languages [Mac86], and deal only essentially with ML-style polymorphism.

Many abstract data types that arise in practice are parameterized: as an example consider the data type of finite sets over a data type that has an operation of equality on it. Semantically such a data type may be regarded as a functor from a category of algebras of a certain signature (parameter algebras), to a category of algebras of a larger signature. An implementation of a parameterized data type thus has to be a functor (as in **XML** [HMM90]) which takes a structure as an argument and returns a structure as result. The study here would involve developing a notion of logical relations for this system. We can then carry out a program similar to the one in this paper, for parameterized data types.

This work is attempting to discover practical verification techniques for the correctness of implementations, and this is a continuing effort. We have indicated one technique, and we have also indicated a sufficient condition to ensure that verifying finitely many observational equivalences is enough, to verify that a given implementation is a valid implementation (we have formalized this notion by the modularity principle). However abstract data types are specified using a finite set of formulae (equations, Horn Clauses etc.) in some specification paradigm, and sufficient conditions are best stated for these finite specifications (as opposed to stating conditions on the algebra itself).

## References

[Abr90] S. Abramsky. Abstract Interpretation, Logical Relations, and Kan Extensions. *J. Logic and Computation*, 1, 1990.

[Ama88] R. M. Amadio. A Fixed Point Extension of the Second-Order Lambda Calculus: Observable Equivalences and Models. In *Proceedings of the Symposium on Logic in Computer Science*, pages 51–60. IEEE, July 1988.

[Don79] J. Donahue. On the Semantics of Data Type. *SIAM J. Computing*, 8:546–560, 1979.

[Fri75] H. Friedman. Equality between Functionals. In R. Parikh, editor, *Proceedings of the Logic Colloqium '73*, pages 22–37. *Lecture Notes in Mathematics*, Vol. 453, Springer-Verlag, 1975.

[Hay84] C. T. Haynes. A Theory of Data Type Representation Independence. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Proceedings of the Conference on Semantics of Data Types, Sophia-Antipolis, June 1984*, pages 157–176. *Lecture Notes in Computer Science*, Vol. 173, Springer-Verlag, 1984.

[HMM90] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order Modules and the Phase Distinction. In *Principles of Programming Languages*, pages 341–354, January 1990.

[Mac86] D. B. MacQueen. Using Dependent Types to Express Modular Structure. In *Conf. Record Thirteenth Ann. Symp. Principles of Programming Languages*, pages 277–286. ACM, January 1986.

[MC88] A. R. Meyer and S. S. Cosmadakis. Semantical Paradigms: Notes for an Invited Lecture. In *Proceedings of the Symposium on Logic in Computer Science*, pages 236–255. IEEE, July 1988.

[Mit86] J. C. Mitchell. Representation Independence and Data Abstraction. In *Proceedings of the 13th Symposium on Principles of Programming Languages*, pages 263–276. ACM, January 1986.

[MM85] J. C. Mitchell and A. R. Meyer. Second-order logical relations (extended abstract). In R. Parikh, editor, *Proceedings of the Conference on Logics of Programs, Brooklyn, June 1985*, pages 225–236. *Lecture Notes in Computer Science*, Vol. 193, Springer-Verlag, 1985.

[MP85] J. C. Mitchell and G. D. Plotkin. Abstract Types have Existential Type. In *Proceedings of the 12th Symposium on Principles of Programming Languages*, pages 37–51. ACM, January 1985.

[Plo77] G. D. Plotkin. LCF Considered as a Programming Language. *Theoretical Computer Science*, 5(3):223–256, December 1977.

[Plo80] G. D. Plotkin. 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.

[Rey74] J. C. Reynolds. Towards a Theory of Type Structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425. *Springer Lecture Notes in Computer Science*, Vol. 19, Springer-Verlag, 1974.

[Rey83] J. C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1983.

[Sta85] R. Statman. Logical Relations and the Typed $\lambda$-calculus. *Information and Control*, 65:85–97, 1985.