DTRE—A Semi-Automatic Transformation System

Lee Blaine and Allen Goldberg

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, California 94304, USA

Abstract

This paper describes the theoretical framework and an implemented system (DTRE) for the specification and verified refinement of specifications using operations on abstract data types. The system is *semi-automatic* in that users can specify some (possibly none) of the implementations and the system will determine the rest of the implementations.

Data types are specified as *parameterized theories* within many-sorted first-order logic; usually these theories are centered around *inductive* sorts. Abstract specifications (theories) are refined in a stepwise fashion into increasingly more concrete theories. Our primary method of refinement is based on *theory interpretation* [1, 2, 3].

Theories and interpretations provide a clean, logically based separation between types and their implementations; thus permitting specification to proceed independently of implementation while simultaneously providing a basis for rapid and verifiably correct transformation to efficient code. DTRE provides a convenient syntax for the description and use of interpretations as refinements. Interpretations can be composed both horizontally (as parameters) and vertically (sequentially) and the DTRE language provides the means for describing implementations via such compositions. Pragmas (program annotations) are written using these terms to direct the compiler to implement particular objects and operations using the specified refinements. Hence, users (including components of the system) can change implementations by simply changing annotations. The DTRE system has been integrated with the KIDS system[4] for algorithm design and refinement.

1 Introduction

The notions of *abstraction*, particularly *data Abstraction*, and *verified refinement* are fundamental to the construction of software systems. Despite the fact that these concepts are to date only weakly supported in languages such as ADA and design tools such as JSD [5], they still have had a significant impact on software productivity.

Experimental systems such as those supporting algebraically-specified abstract types [6] provide axiomatic specification of semantics—albeit in the restricted language of equational logic—but do not support refinement and rely on weak methods of implementation, namely term rewriting. While term rewriting is a largely automatic method suitable for prototyping, we do not believe that it represents a viable methodology for the development of production-quality systems.

In this paper we present the Data Type REfinement system (DTRE) which supports the abstract specification and verified refinement of axiomatically-specified data types. The system is geared toward the construction of production-quality code. The system has been used to refine set-theoretic abstract types (sets, sequences, finite maps, and tuples) into diverse lower-level implementations, such as lists, bit-vectors, and array-based representations. The refinements themselves have been proven correct, some with the aid of a mechanical theorem prover.

Section 2 of this paper describes the theoretical foundations for this work. Data types are specified as *parameterized theories* within many-sorted first-order logic; usually these theories are centered around *inductive* sorts. Abstract specifications (theories) are refined in a stepwise fashion into a sequence of increasingly more concrete theories. The general notion of correct refinement is the same as that of Sannella and Tarlecki in [7, 8]: the specification S' is a refinement of S iff every model of S' is a model of S. Our primary method of refinement is based on *theory interpretation* [1, 2, 3]. Roughly speaking, given two theories, $\mathcal{T}_1$ and $\mathcal{T}_2$ a theory interpretation of $\mathcal{T}_1$ into $\mathcal{T}_2$ is a translation mapping expressions written in the language of $\mathcal{T}_1$ into expressions in $\mathcal{T}_2$ such that the translation of the axioms of $\mathcal{T}_1$ are provable from the axioms of $\mathcal{T}_2$. An interpretation of $\mathcal{T}_1$ into $\mathcal{T}_2$ ensures the existence of a construction transforming models of $\mathcal{T}_2$ into models of $\mathcal{T}_1$; hence, establishing the correctness of the refinement. Correctness of interpretations is established once, not at each use; however, there are often conditions on the use of a particular interpretation for a given refinement that must be verified for each use. For example, the partial implementation (only some operations are implemented) of sets as stacks requires establishing certain data flow relations.

Theories and interpretations provide a clean, logically-based separation between types and their implementations; thus permitting specification to proceed independently of implementation while simultaneously providing a basis for rapid and verifiably correct transformation to efficient code.

Section 3 describes the DTRE system, a semi-automatic synthesis system designed to utilize implementation directives to synthesize efficient code from high-level specifications in a verifiably correct manner. DTRE provides a convenient syntax for the description and use of interpretations as refinements. Interpretations can be composed both horizontally (as parameters) and vertically (sequentially) and the DTRE language provides the means for describing implementations via such compositions. Pragmas (program annotations) are written using these terms to direct the compiler to implement particular objects and operations using the specified refinements. Hence, users can change implementations by simply changing annotations.

Operationally, interpretations from a theory $\mathcal{T}_1$ to a theory $\mathcal{T}_2$ are compiled into collections of transformation rules which when applied exhaustively to a program written using the types and operations of theory $\mathcal{T}_1$ transform the program into one over the types and operations of theory $\mathcal{T}_2$. The current implementation contains a library of refinements which can be used to implement set-theoretic types (sets, maps, sequences, and tuples) in terms of arrays, lists, bit-vectors, etc.

We believe that the primary advantages of DTRE are those stemming from its foundation in theories and interpretations between theories: the incremental capture and capitalization of knowledge and its utilization of that foundation as a basis for rapid and verifiably correct transformation to efficient code. DTRE's primary disadvantages are those stemming from its immaturity and incompleteness. I.e., too few theories and interpretations; and, in general, the methods for adding user specified theories and interpretations currently are in a primitive state.

DTRE has a number of capabilities that are utilized in attempting to generate efficient code (some of which are beyond the scope of this paper to describe in detail). Some of those capabilities are:

Partial Implementations—DTRE supports partial implementations [9], i.e. implementations of a type that support a subset of the defined operations. For example, DTRE can use a stack to represent a set, provided that only certain operations are performed on the set and that certain data flow conditions are met.

Multiple Representations and Conversions—Objects involved in a single operation can have differing implementations. For example, in $A \cup B$, A and B can have the same or different implementations; further, the implementation of the result can be the same as either or both arguments or it can be yet another implementation. Correct code can be generated for any of the possibilities. In cases where objects must have the same representation (e.g., they are both elements of a set literal), the system is designed to generate conversion code.

Multiple Targets—DTRE is designed to support the generation of code in multiple languages. An abstract target language at the level of arrays, pointers, and records is defined and abstract types are refined down to this level. Additional refinement steps map from the abstract target language to particular programming languages like COMMON LISP, C or ADA. The system is also designed to compose the two steps and map directly to the target language.

In-Place Modification and Copy Optimization—Many useful abstract types are large composite objects which are expensive to copy. Naive translation often introduces unnecessary copy operations. DTRE uses data flow and value flow analysis to determine when destructive operations may replace copy operations.

Proceduralization—To reduce the size of the generated code DTRE has the capability to proceduralize its implementation of abstract operations. (In essence, this is lambda abstraction with implementation directives.)

Program Analysis—The system utilizes such information as data flow and value flow analysis, operations analysis (the set of operations performed on the object), and symbolic containment and size bounds on objects in attempting to select efficient implementations. Planned upgrades include estimating the relative efficiency of operations on given implementations [10]. Currently, for efficiency reasons, the theorem proving utilized during data structure selection and transformation is rather shallow: essentially fact gathering augmented by computing closures. The theorem proving involved in establishing interpretations is somewhat deeper but still routine.

Data Structure Selection—DTRE utilizes a semi-automatic Data Structure Selection (DSS) system: the user provides some implementation directives (possibly none) and the system determines the rest. It utilizes the information from the program analysis along with the choices already made in selecting implementations for those objects whose implementations have not been determined.

Section 4 describes related work.

2 Theories, Interpretations, and Refinement

This section describes the logical framework on which DTRE is based. We describe and give examples of such basic notions as: theory, theory interpretation, refinement, and implementation. The primary example is an interpretation of $\texttt{Set}(\alpha)$ (the theory of sets whose elements are of type α) in $\texttt{List}(\beta)$ (the theory of lists whose elements are of type β). To this end we give descriptions of the theories $\texttt{Set}(\alpha)$ and $\texttt{List}(\beta)$. The notation used in these examples is a more readable notation than that used by the DTRE system.

2.1 Theories

In DTRE generic data types are formulated as *parameterized theories* within many-sorted first-order logic; usually these theories are centered around *inductive* sorts. Axioms may be unrestricted first-order formulas or schemas. A special notation is used for specifying inductive sorts.

Each theory has a **signature** $\Sigma = \langle S, \Omega\rangle$ where S is a set of sort names and Ω is a family of sets $\{\Omega_{w,s}\}_{w\in S^*, s\in S}$ of operation names indexed by *arity* and result sorts. A theory presentation is a pair $\langle\Sigma, \Phi\rangle$ where Σ is a signature giving the vocabulary of the theory and Φ is a set of axioms for the theory. The theory denoted by $\langle\Sigma, \Phi\rangle$ is the set of consequences of Φ[1].

A concrete *theory presentation* can have the following components:

[1] In general the abstract notation is that employed in the algebraic specification literature. E.g., see [11, 8].

Name—A name and parameters, used to identify the theory. The parameters are used to represent sorts of the theory that have yet to be instantiated. DTRE actually supports a more sophisticated notion of parameterized theory in which full theories are passed as parameters, but a description of this is beyond the scope of this paper. Since all theories are assumed to have an equality predicate, reference to the equality relation of a parameter is valid within a theory definition.

Imports—A list of imported theories. The types, operations, and axioms of the theory are the union of the types, operations, and axioms defined in the imported theories and those defined explicitly.

Types—A list of base types and type constructors. These introduce new types and generic type constructors. Type constructors are n-ary functions from types to types. The sort parameters of the theory are associated with the parameters of the type constructors to achieve generic types. We use Greek letters to range over type variables. For example, a theory may define type constructors $set(\alpha)$, and $seq(\beta)$. The only assumption made on the sorts α and β is that equality is defined.

Operations—A collection of operation signatures. This introduces the operations of the theory, i.e. functions, relations and constants defined by the theory. The signatures of generic operations are defined using type variables. For example, the signature of the *image* operation of a set by a map is $image : map(\alpha, \beta) \times set(\alpha) \rightarrow set(\beta)$. The signature of the generic empty set constant is $\{\,\} : set(\alpha)$

Axioms—A collection of axioms and axiom schemas. Axioms are written using standard first order notation with variables over the generic sorts. For example, an axiom asserting the commutativity of a set union operation can be written

$$\forall(S_1 : \texttt{Set}(\alpha))\ \forall(S_2 : \texttt{Set}(\alpha))\ S_1 \cup S_2 = S_2 \cup S_1$$

Initial universal quantifiers will be omitted when convenient as will type information that can be inferred by a ML-style type inference procedure. First order axiom schemas indexed by formulas $\phi(\bar{x})$ over the language of the theory are permitted. A special form of axiom schema which facilitates the definition of inductive theories is provided. Specifically the notation:

constructors $\{c_1, \ldots, c_k, f_1, \ldots, f_m\}$ **construct** τ

where each c_i is a constant of type τ and each f_i is an n_i-ary function with signature $\tau \times s_1 \times \ldots, \times s_{n_i} \rightarrow \tau$ — with each of the s_i distinct from τ [2] asserts that every object of type τ can be finitely generated from the constants $c_1, \ldots, c_k$ and the operations $f_1, \ldots, f_m$. A use of the **constructors** notation introduces the corresponding induction schema:

[2]This restriction is made to simplify notation.

$$\phi(c_1) \wedge \ldots \wedge \phi(c_k) \ \wedge \ \bigwedge_{i=1}^{m} \forall(x:\tau)[\forall(\bar{y}:\bar{s})\ \phi(x) \Rightarrow \phi(f_i(x,\bar{y}))] \Rightarrow \forall(x:\tau)\ \phi(x)$$

where ϕ is a formula in which the $\bar{y}$ do not occur free [12, 13]. The use of the **constructors** notation also enables the use of definition by *structural induction.*

Axiom schemas, i.e. formulas with terms and formulas as parameters are also permitted.

Theory Development—This is a sequence of definitions and theorems (the dependency order is a tree). Generally to keep the specification of theory interpretations concise, the number of constant, function, and relation symbols defined in a theory is kept to a minimum. Additional operations are introduced by definitions, which yield conservative extensions to the theory. Theorems are stated and latter referred to to prove that theory interpretations are correct, and to facilitate reasoning about programs written using the theory. Our presentation of theories, particularly the theory development part will be interspersed with explanations.

A *structure* for an unparameterized theory is defined in the usual way: it consists of a collection of non-empty sets called *domains* for each sort of the theory together with operations (i.e. functions, relations and constants) over the domains corresponding to the operations of the theory.

A *structure* for a theory parameterized by sorts $\alpha_1 \ldots \alpha_N$ is a function that when given a structure (i.e. a set with equality relation over the set) for each parameter yields a structure for the theory.

A *model* of a theory is a structure in which all of the axioms are true. $Mod(\mathcal{T})$ denotes the set of all models of $\mathcal{T}$.

2.1.1 Theories with Equality

For each sort of a theory there is an equality relation denoted by the (overloaded) symbol $=$. Where confusion may result we subscript the equality symbol with the sort to which it applies. When presenting a theory we will not explicitly mention equality or its associated axioms. However it is understood that the usual axioms for equality hold: the axioms that assert that equality is an equivalence relation, and that for each *n-ary* function symbol f (or relation symbol) of the theory there is an equality substitution axiom

$$\vdash t_1 = t_1' \wedge \ldots \wedge t_n = t_n' \Rightarrow f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')$$

I.e., equality is a congruence relation in the theory. When a theory interpretation is given it is often not the case that equality of a sort α is mapped to equality of the sort that interprets α. For example, when the theory of sets is interpreted in the theory of lists, equality on sets is interpreted by a defined relation not by equality on lists. Thus, to prove that a theory interpretation is correct it is necessary to show that the interpretations of the axioms of equality are derivable in the target theory.

2.2 Theory Extensions

Extensions of theories play a major role both theoretically and practically in the refinement of specifications to code. In this section we review some of the logical issues involved in theory extensions.

A theory is a set of theorems closed under consequence; hence, proving a theorem does not extend the theory, merely our knowledge of the theory or its presentation. However, adding axioms that are not already theorems of the theory does extend the theory. Also, definitions extend both the language of the theory and the theory itself by adding new symbols to the language and by adding the defining formulas as new axioms to the theory. On the other hand, definitional extensions should be merely linguistic and not creative: replacing defined symbols by their definitions should yield a theory equivalent to the original theory. In the sections below we review some of the basic notions and results for various forms of definitional extensions. See [14, 1, 2] for these results and the details.

A theory $\mathcal{T}_2$ is an *extension* of a theory $\mathcal{T}_1$ ($\mathcal{T}_1 \subseteq \mathcal{T}_2$) iff $\mathcal{L}_2$ is an extension of $\mathcal{L}_1$ and every theorem of $\mathcal{T}_1$ is a theorem of $\mathcal{T}_2$. $\mathcal{T}_2$ is a *correct* extension iff the consistency of $\mathcal{T}_1$ implies the consistency of $\mathcal{T}_2$. $\mathcal{T}_2$ is a *conservative* extension of $\mathcal{T}_1$ iff it is correct extension of $\mathcal{T}_1$ and for every formula ϕ of $\mathcal{L}_1$, if $\vdash_{\mathcal{T}_2} \phi$ then $\vdash_{\mathcal{T}_1} \phi$.

A definition of symbol ζ will be called *explicit* if ζ does not occur in the right hand side of the definition. Otherwise, the definition is *implicit*; usually these are *recursive* or *inductive* definitions.

2.2.1 Definitional Extensions

A theory $\mathcal{T}_2$ is a *definitional extension* of a theory $\mathcal{T}_1$ iff $\mathcal{L}_2$ is obtained from $\mathcal{L}_1$ by introducing a new (function, relation or constant) symbol and the axioms of $\mathcal{T}_2$ are the axioms of $\mathcal{T}_1$ plus a definition (set of *defining axioms*). A definition *defines* (picks out) an object whose existence is a consequence of the current theory; hence, a definitional extension based on a proper definition is consistent if the base theory is consistent.

The following sections deal primarily with the case of extensions that add new functions; the case of relations is simpler. Explicit definitions of functions are of the form [3]:

$$f(x) = y \;\Leftrightarrow\; \phi[x, y]$$

[3]The extension to the case of conditional definitions is straightforward

where f does not appear in $\phi[x, y]$. For such definitions there are two conditions that ensure that they are legitimate [4]:

Existence—The definition should define a function in the current theory: it should pick out some function whose pointwise existence and uniqueness is a consequence of the current theory. Hence, if f is new function symbol and $\mathcal{T}_1$ is the current theory, then it must be the case that:

$$\mathcal{T}_1 \ \vdash \forall(x)\, \exists!(y)\, \phi[x, y]$$

Conservative Extension—The definition should not be creative: the extended theory should be a *conservative extension* of the current theory. If $\mathcal{T}_1$ is the current theory, $\mathcal{T}_2$ is the extended theory, and ϕ is a formula in the language of $\mathcal{T}_1$, then

$$\vdash_{\mathcal{T}_2} \phi \quad \Rightarrow \ \vdash_{\mathcal{T}_1} \phi$$

We will say that a definitional extension is *justified* iff the definition defines a function and the extension is a conservative extension. For explicit definitions, satisfaction of the existence condition implies that the definitional extension is a conservative extension[1, 2]. The basic form of the argument is as follows, where $\mathcal{T}_1$ is the original theory and $\mathcal{T}_2$ is $\mathcal{T}_1$ extended with the defining axioms for the function symbol f.

Existence—Provability of the pointwise existence and uniqueness condition in $\mathcal{T}_1$ implies that there exists a function F satisfying $\mathcal{T}_1$ plus the defining axioms.

Conservative Extension—Existence of a function F satisfying $\mathcal{T}_1$ plus the defining axioms implies that $\mathcal{T}_2$ is a conservative extension of $\mathcal{T}_1$. To see this, let $\mathcal{M}_1$ be a model of $\mathcal{T}_1$. Then $\mathcal{M}_1$ extended by F is a model of $\mathcal{T}_2$. Hence, $\mathcal{T}_2$ is a correct extension of $\mathcal{T}_1$. Further, if ϕ is a formula of $\mathcal{L}_1$ (ϕ does not mention F), then $\models_{\mathcal{M}_2} \phi \quad \Rightarrow \ \models_{\mathcal{M}_1} \phi$. It follows that $\mathcal{T}_2$ is a conservative extension of $\mathcal{T}_1$

Hence, for explicit definitions, the question of justification reduces to the question of existence; which, in turn, reduces to the pointwise existence and uniqueness condition. We will often use 'justification condition' to refer to this residue condition that must be established in each case.

2.2.2 Extensions by Implicit Definitions

A recursive definition of a function is one in which the defining axiom of the function is of the form

$$f(x) = y \ \Leftrightarrow \ \phi[x, y]$$

[4] There are other conditions on free variables, etc. See [14, 1, 2] for the full details

where $\phi[x, y]$ does involve occurrences of f. For recursive definitions, the situation is similar to that of explicit definitions: if the existence of a function satisfying the defining axioms can be established, then it follows that the definitional extension is conservative extension. Hence, what must be established is the existence of a function satisfying the defining axioms.

We review some standard methods of demonstrating the existence of a function (not necessarily unique) that satisfies an implicit definition.

The most commonly encountered method of inductive definition is the use of structural induction over the generators of a sort. Recall that the notation

constructors $\{c_1, \ldots, c_k, f_1, \ldots, f_m\}$ **construct** τ

asserts an induction principle which informally states that every object of type τ can be finitely generated from the constants $c_1, \ldots, c_k$ and the operations $f_1, \ldots, f_m$.

Suppose a function f is defined by an axiom schema of the form

$$\begin{array}{rcl} f(c_i, \vec{p}) & = & g_i(\vec{p}) \\ f(f_i(x, \vec{y}), \vec{p})) & = & H(f(x, \vec{p}), x, \vec{y}, \vec{p})) \end{array}$$

Such a definition is said to be defined by structural induction over the constructors of τ. By using the induction schema it is straightforward to prove $\forall(x)\phi(x)$ where $\phi(x)$ is:

$$\exists(y)\ y = f(x, \vec{p})$$

which asserts the existence part of the justification condition. The proof of the uniqueness of such a y can not be demonstrated in general. The theory may have axioms which assert the equality of syntactically distinct terms of the type τ; e.g., if τ is not freely generated from its constructors. A function definition by structural induction defines a value for each term. If the values given for syntactically distinct, but provably equal terms are different the function "definition" does not actually define a function. Thus, when a function definition is introduced a proof that the function is well-defined must be given. An example is given in the next section. In this paper all inductive definitions will be based on structural induction over constructors.

More general methods of establishing the existence of functions satisfying recursive definitions are based on the theory of fixed points of functionals. For example, see [15].

2.3 Theory of $\texttt{Set}(\alpha)$

In this section we give an example of a theory, the theory of $\texttt{Set}(\alpha)$: the theory of finite sets whose elements are of type α.

The theory is a parameterized first-order theory based on $\texttt{Set}$, $=$, $\{\,\}$, $\texttt{with}$, and $\in$. All other notions are defined, usually by induction over $\{\,\}$ and $\texttt{with}$. It is assumed that equality is available on the parameter type. Relations are typed as boolean valued maps. Only the simplest form of (separation) comprehension is presented: the form with multiple base sets and with extra parameters $\vec{p}$ is valid, but not formalized here. For readability we use some notation that is not part of the standard system.

theory $\texttt{Set}(\alpha)$

imports $\texttt{integer}$, $\texttt{boolean}$
types α, $\texttt{Set}(\alpha)$
operations
$\{\,\} : \texttt{Set}(\alpha)$
$\texttt{with} : \texttt{Set}(\alpha) \times \alpha \rightarrow \texttt{Set}(\alpha)$
$\in : \alpha \times \texttt{Set}(\alpha) \rightarrow \texttt{boolean}$

axioms
`membership`
$x \notin \{\,\}$
$x \in (S \texttt{ with } y) \Leftrightarrow x = y \vee x \in S$

`with`
The *interchange* axiom for `with`
$(S \texttt{ with } x) \texttt{ with } y = (S \texttt{ with } y) \texttt{ with } x$
The *idempotence* axiom for `with`
$(S \texttt{ with } x) \texttt{ with } x = S \texttt{ with } x$

`set-equality`
The equivalence axioms for equality on sets.
The substitution axioms for equality on sets.

constructors $\{\{\,\}, \texttt{ with }\}$ **construct** $\texttt{Set}(\alpha)$

The constructor clause for sets introduces the induction schema:
If ϕ is a formula in which x does not occur free, then
$$\phi[\{\,\}] \wedge \forall(S,x)(\phi[S] \Rightarrow \phi[S \texttt{ with } x]) \Rightarrow \forall(S)\phi[S]$$

We have defined sets using a very small set of operations and axioms. In the theory development section we shall extend the theory. Our development is similar to that in [12].

theory development

definition: `empty?`

$\texttt{empty?}(S) \Leftrightarrow S = \{\,\}$

theorem: `finite constructability`

$S = \{\,\} \vee \exists(x)\, \exists(S' : \texttt{Set}(\alpha))(S = S' \texttt{ with } x \ \wedge \ x \notin S')$

theorem: `extensionality`

$S_1 = S_2 \Leftrightarrow \forall(x)(x \in S_1 \Leftrightarrow x \in S_2)$

These two theorems state basic properties of sets. The first states that every set can be written in the form

$$(\ldots(\{\,\} \texttt{ with } t_1)\ldots) \texttt{ with } t_n$$

where $i \neq j \Rightarrow \vdash_\alpha t_i \neq t_j$. The second asserts that two sets are equal if and only if they have the same members. Each of these theorems can be proven by induction over the constructor set. See [12] for details.

A very powerful notation for defining sets is the set former or comprehension operator which permits a set to be defined by providing a description of its elements. Set formers are defined and axiomatized by an axiom scheme in which instances are formed by instantiating a term $\mathbf{Tm}[x]$ and a formula $\mathbf{Fm}[x]$ each with a single free variable x of type α.

Set Formers

For $S : \texttt{Set}(\alpha)$, formula $\mathbf{Fm}[x]$ and term $\mathbf{Tm}[x] : \ \alpha \to \alpha$
the set comprehension $\{\mathbf{Tm}(x) \mid (x \in S)\ \mathbf{Fm}[x]\}$
denotes a function of type: $\texttt{Set}(\alpha) \to \texttt{Set}(\alpha)$

$$\{\mathbf{Tm}(x) \mid (x \in S)\ \mathbf{Fm}[x]\}$$
$$= \begin{cases} \{\,\} & \text{if } S = \{\,\} \\ \{\mathbf{Tm}(x) \mid (x \in S')\ \mathbf{Fm}[x]\} & \text{if } S = S' \texttt{ with } a \wedge \neg\mathbf{Fm}[a] \\ \{\mathbf{Tm}(x) \mid (x \in S')\ \mathbf{Fm}[x]\} \texttt{ with } \mathbf{Tm}[a] & \text{if } S = S' \texttt{ with } a \wedge \mathbf{Fm}[a] \ \wedge \ a \notin S' \end{cases}$$

This is an inductive definition of the set former operation. From it we can prove:

theorem: `separation`

$$\forall(z)\ (z \in \{\mathbf{Tm}[x] \mid (x \in S)\ \mathbf{Fm}[x]\} \Leftrightarrow \exists(x)\ (x \in S \wedge \mathbf{Fm}[x] \wedge z = \mathbf{Tm}[x])$$

definition: $\subseteq$

$\{\,\} \subseteq S_2$
$(S \ \mathtt{with}\ x) \subseteq S_2 \Leftrightarrow S \subseteq S_2 \wedge x \in S_2$

theorem: $\subseteq$

$S_1 \subseteq S_2 \Leftrightarrow \forall(x \in S_1) x \in S_2$

definition: `size`

$size(\{\,\}) = 0$
$size(S \ \mathtt{with}\ x) = (\mathtt{if}\ x \in S\ \mathtt{then}\ size(S)\ \mathtt{else}\ size(S) + 1)$

theorem: `set-equality`

$S_1 = S_2 \Leftrightarrow \mathtt{size}(S_1) = \mathtt{size}(S_2) \wedge \forall(x)(x \in S_1 \Leftrightarrow x \in S_2)$

definition: $\cup$

$S_1 \cup \{\,\} = S_1$
$S_1 \cup (S\ \mathtt{with}\ x) = (S_1 \cup S)\ \mathtt{with}\ x$

theorem: $\cup$

$x \in S_1 \cup S_2 \Leftrightarrow x \in S_1 \vee x \in S_2$

definition: $\cap$

$S_1 \cap \{\,\} = \{\,\}$
$S_1 \cap (S\ \mathtt{with}\ x) = (\mathtt{if}\ x \in S_1\ \mathtt{then}\ (S_1 \cap S)\ \mathtt{with}\ x\ \mathtt{else}\ (S_1 \cap S))$

theorem: $\cap$

$S_1 \cap S_2 = \{x \mid (x \in S_1)\ x \in S_2\} = \{x \mid (x \in S_2)\ x \in S_1\}$
$x \in S_1 \cap S_2 \Leftrightarrow x \in S_1 \wedge x \in S_2$

definition: `less`

$\{\,\}\ \mathtt{less}\ y = \{\,\}$
$(S\ \mathtt{with}\ x)\ \mathtt{less}\ y = (\mathtt{if}\ x = y\ \mathtt{then}\ S\ \mathtt{less}\ y\ \mathtt{else}\ (S\ \mathtt{less}\ y)\ \mathtt{with}\ x)$

theorem: `less`

$S\ \mathtt{less}\ y = \{x \mid (x \in S)\ x \neq y\}$
$x \in S\ \mathtt{less}\ y \Leftrightarrow x \in S \wedge \neg(x = y)$
$x \notin S \Rightarrow (S\ \mathtt{with}\ x)\ \mathtt{less}\ y = (\mathtt{if}\ x = y\ \mathtt{then}\ S\ \mathtt{else}\ (S\ \mathtt{less}\ y)\ \mathtt{with}\ x)$

definition: $\texttt{setdiff}$

$\texttt{setdiff}(\{\,\}, S_2) = \{\,\}$
$\texttt{setdiff}(S \texttt{ with } x, S_2) =$
$(\texttt{if } x \in S_2 \texttt{ then setdiff}(S, S_2) \texttt{ else setdiff}(S, S_2) \texttt{ with } x)$

theorem: $\texttt{setdiff}$

$\texttt{setdiff}(S_1, S_2) = \{x \mid (x \in S_1) \neg x \in S_2\}$
$x \in \texttt{setdiff}(S_1, S_2) \Leftrightarrow x \in S_1 \wedge \neg x \in S_2$

end-theory

2.4 Definitional Extensions of $\texttt{Set}(\alpha)$

Many of the definitional extensions above, such as, $\cup$, $\texttt{size}$, $\cap$, $\texttt{less}$, $\texttt{setdiff}$, etc. are defined by structural induction inductive, based on the constructor set $\{\{\,\}, \texttt{ with }\}$. We must show that these definitions define functions, and hence by the discussion in Section2.2, induce conservative extensions. To illustrate the potential problem consider the "definition" of a function $\texttt{goofy} : \texttt{Set}(\alpha) \to \alpha$ by the following induction:

$\texttt{goofy}(\{\,\}) = c_\alpha$
$\texttt{goofy}(S \texttt{ with } x) = x$

Here c_α denotes an arbitrary element of type α. By definition, $\texttt{goofy}((S \texttt{ with } x) \texttt{ with } y) = y$. But by the interchange axiom and the substitution axiom for equality,

$$\texttt{goofy}((S \texttt{ with } x) \texttt{ with } y) = \texttt{goofy}((S \texttt{ with } y) \texttt{ with } x) = x$$

implying $\forall(x, y)\, x = y$. This is an inconsistency since it requires that models of α contain at most one individual.

The problem is that the axioms defining $\texttt{goofy}$ do not imply that for provably equal terms t_1, t_2 of set theory, $\texttt{goofy}(t_1) = \texttt{goofy}(t_2)$, which is the substitution axiom for equality. The equivalence relation on terms induced by equality in $\texttt{Set}(\alpha)$ is not a congruence relation with respect to the $\texttt{goofy}$ function. In other words, the choice of representatives matters.

Hence, what must be shown to justify a definition for the function f defined by structural induction on the constructors is equality substitution (congruence) for f: if t_1 and t_2 are equal in $\texttt{Set}(\alpha)$ then $f(t_1) = f(t_2)$. The following theorem gives a simple-to-verify condition that establishes congruence for f; namely, that the terms in the interchange and idempotence axioms are congruent with respect to f.

Theorem: If $f(S, \vec{p})$ is defined by an inductive definition of the form:

$$f(S,\vec{p}) = \begin{cases} g(\vec{p}) & \text{if } S = \{\,\} \\ H(f(S',\vec{p}),S,x,\vec{p}) & \text{if } S = S' \texttt{ with } x \end{cases}$$

and the following conditions hold:

1. $\vdash_{Setp(\alpha)} S = T$
2. $\vdash_{Setp(\alpha)} f((S \texttt{ with } t_1) \texttt{ with } t_1, \vec{p}) = f(S \texttt{ with } t_1, \vec{p})$
3. $\vdash_{Setp(\alpha)} f((S \texttt{ with } t_1) \texttt{ with } t_2, \vec{p}) = f((S \texttt{ with } t_2) \texttt{ with } t_1, \vec{p})$

then $\vdash_{Setp(\alpha)} f(S,\vec{p}) = f(T,\vec{p})$

For example, to show that $\cup$ is a well-defined function:

$$\begin{aligned} S_1 \cup ((S \texttt{ with } x) \texttt{ with } y) &= (S_1 \cup (S \texttt{ with } x)) \texttt{ with } y \\ &= ((S_1 \cup S) \texttt{ with } x) \texttt{ with } y \\ &= ((S_1 \cup S) \texttt{ with } y) \texttt{ with } x \\ &= (S_1 \cup (S \texttt{ with } y)) \texttt{ with } x \\ &= S_1 \cup ((S \texttt{ with } y) \texttt{ with } x) \end{aligned}$$

and

$$\begin{aligned} S_1 \cup ((S \texttt{ with } x) \texttt{ with } x) &= (S_1 \cup (S \texttt{ with } x)) \texttt{ with } x \\ &= ((S_1 \cup S) \texttt{ with } x) \texttt{ with } x \\ &= (S_1 \cup S) \texttt{ with } x \\ &= S_1 \cup (S \texttt{ with } x) \end{aligned}$$

Proofs of the other definitions are similar. It is desirable to seek a syntactic condition that is sufficient to establish for a given definition of $f(S,\vec{p})$, whether or not the definition satisfies congruence with respect to the axioms of interchange and idempotence. Unfortunately, we do not have a definitive test for this at the moment. What empirical data show is that the proofs all have the same simple form: use the definition to push the defined symbol down as far as possible; use interchange, idempotence, and simple logic to put the reduced form into DNF and to manipulate the DNF to the DNF basis for the result; and finally use the definition to lift the defined symbol back to the top. If the definition does not use conditionals, then the DNF is trivial: it is simply the single term. Hence, it should be a simple task for a resolution based theorem prover to decide.

2.5 Theory of $\texttt{List}(\beta)$

The theory is a parameterized first order theory based on `List`, $=$, `nil`, `cons`, `head` and `tail`. All other notions are defined, usually by induction over `nil` and `cons`. It is assumed that equality is available on the parameter type. We will not give an extensive axiomatization of list, but simply provide a basis for proving the interpretation of set theory in list theory. Where proofs are given, the end of the proof is indicated by $\Box$.

theory $\texttt{List}(\beta)$
imports `integer`, `boolean`
types β, $\texttt{List}(\beta)$
operations
  `nil`: $\texttt{List}(\beta)$
  `cons`: $\beta \times \texttt{List}(\beta) \rightarrow \texttt{List}(\beta)$
  `head`: $\texttt{List}(\beta) \rightarrow \beta$
  `tail`: $\texttt{List}(\beta) \rightarrow \texttt{List}(\beta)$

axioms
`cons`
  $\texttt{nil} \neq \texttt{cons}(x, L)$
  $\texttt{cons}(x, L_1) = \texttt{cons}(y, L_2) \Leftrightarrow x = y \wedge L_1 = L_2$

`head`
  $\texttt{head}(\texttt{cons}(x, L)) = x$

`tail`
  $\texttt{tail}(\texttt{cons}(x, L)) = L$

`list-equality`
  The equivalence axioms for equality on lists.
  The substitution axioms for equality on lists.

constructors {`nil`, `cons`} **construct** `List`

The constructor clause for lists introduces the induction schema:
  If ϕ is a formula then
    $\phi[\texttt{nil}] \wedge \forall(L, x)(\phi[L] \Rightarrow \phi[\texttt{cons}(x, L)]) \Rightarrow \forall(L)\phi[L]$

theory development

definition: `null?`
  $\texttt{null?}(L) \Leftrightarrow L = \texttt{nil}$

General List Formers
For $L : \texttt{List}(\beta)$, formula $\mathbf{Fm}[x]$ and term $\mathbf{Tm}[x] : \ \beta \to \beta$
the list comprehension $\quad [\mathbf{Tm}(x) \mid (x \in L)\ \mathbf{Fm}[x]]$
denotes a function of type: $\texttt{List}(\beta) \to \texttt{List}(\beta)$

$$[\mathbf{Tm}(x) \mid (x \in L)\ \mathbf{Fm}[x]] = \begin{cases} [\,] & \text{if } L = \texttt{nil} \\ [\mathbf{Tm}(x) \mid (x \in L')\ \mathbf{Fm}[x]] & \text{if } L = \texttt{cons}(a, L') \wedge \neg\mathbf{Fm}[a] \\ \texttt{cons}(\mathbf{Tm}[a], [\mathbf{Tm}(x) \mid (x \in L')\ \mathbf{Fm}[x]]) & \text{if } L = \texttt{cons}(a, L') \wedge \mathbf{Fm}[a] \end{cases}$$

definition: `length`

$$\texttt{length}(L) = \begin{cases} 0 & \text{if } L = \texttt{nil} \\ \texttt{length}(L') + 1 & \text{if } L = \texttt{cons}(x, L') \end{cases}$$

definition: `member?`

$\neg\texttt{member?}(x, \texttt{nil})$

$\texttt{member?}(x, \texttt{cons}(y, L)) \Leftrightarrow x = y \vee \texttt{member?}(x, L)$

definition: `insert`

$$\texttt{insert}(x, L) = \begin{cases} L & \text{if } \texttt{member?}(x, L) \\ \texttt{cons}(x, L) & \text{if } \neg\texttt{member?}(x, L) \end{cases}$$

corollary:

$\texttt{insert}(x, \texttt{nil}) = \texttt{cons}(x, \texttt{nil})$

$\texttt{insert}(x, \texttt{cons}(y, L))$
$\quad = (\texttt{if } x = y \vee \ \texttt{member?}(x, L)$
$\qquad \texttt{then cons}(y, L) \texttt{ else cons}(x, \texttt{cons}(y, L))$

theorem: $\texttt{member?}(x, \texttt{insert}(y, L)) \ \Leftrightarrow x = y \vee \texttt{member?}(x, L)$
proof
$\Rightarrow$: Assume $\texttt{member?}(x, \texttt{insert}(y, L))$
Case 1: $\texttt{member?}(y, L)$
Then $\texttt{insert}(y, L) = L$ Hence, $\texttt{member?}(x, L)$.
Case 2: $\neg\texttt{member?}(y, L)$
Then $\texttt{insert}(y, L) = \texttt{cons}(y, L)$ By definition of `member?`,
$\texttt{member?}(x, \texttt{cons}(y, L))$ => $x = y \vee \texttt{member?}(x, L)$

$\Leftarrow$: Assume $x = y \vee \texttt{member?}(x, L)$
Case 1: $\texttt{member?}(y, L)$
Then $\texttt{insert}(y, L) = L$. If $x = y$ then $\texttt{member?}(x, L)$;
hence, $\texttt{member?}(x, \texttt{insert}(y, L))$.
Case 2: $\neg\texttt{member?}(y, L)$
Then $\texttt{insert}(y, L) = \texttt{cons}(y, L)$ By definition of `member?`,
$x = y \vee \texttt{member?}(x, L)$ => $\texttt{member?}(x, \texttt{cons}(y, L))$
$\square$

corollary: $\texttt{member?}(x, \texttt{insert}(x, L))$

corollary: $\texttt{insert}(x, \texttt{insert}(x, L)) = \texttt{insert}(x, L)$
proof:
By the first corollary, $\texttt{member?}(x, \texttt{insert}(x, L))$. Hence, by the definition of insert, $\texttt{insert}(x, \texttt{insert}(x, L)) = \texttt{insert}(x, L)$ □

theorem: $\texttt{member?}(z, \texttt{insert}(y, \texttt{insert}(x, L))) \Rightarrow \texttt{member?}(z, \texttt{insert}(x, \texttt{insert}(y, L)))$

definition: no-dup?

$\texttt{no-dup?}(\texttt{nil})$

$\texttt{no-dup?}(\texttt{cons}(x, L)) \Leftrightarrow \texttt{no-dup?}(L) \wedge \neg\texttt{member?}(x, L)$

theorem: $\texttt{no-dup?}(L) \Rightarrow \texttt{no-dup?}(\texttt{insert}(x, L))$
proof: By induction on L.
$L = \texttt{nil}$
Because $\neg\texttt{member?}(x, \texttt{nil})$ then $\texttt{insert}(x, \texttt{nil}) = \texttt{cons}(x, \texttt{nil})$
By the definition of no-dup? $\texttt{no-dup?}(\texttt{nil})$; hence, $\texttt{no-dup?}(\texttt{cons}(x, \texttt{nil}))$.
$L = \texttt{cons}(y, L')$
Either $\texttt{member?}(x, \texttt{cons}(y, L')) \vee \neg\texttt{member?}(x, \texttt{cons}(y, L'))$
In the former case, $\texttt{insert}(x, \texttt{cons}(y, L')) = \texttt{cons}(y, L')$; hence, $\texttt{no-dup?}(\texttt{insert}(x, \texttt{cons}(y, L')))$.
In the latter case, $\texttt{insert}(x, \texttt{cons}(y, L')) = \texttt{cons}(x, \texttt{cons}(y, L'))$ and the result is immediate from the definition. □

definition: set-equal?

$\texttt{no-dup?}(L_1) \wedge \texttt{no-dup?}(L_2) \Rightarrow$
$\quad \texttt{set-equal?}(L_1, L_2) \Leftrightarrow \forall(x)(\texttt{member?}(x, L_1) \Leftrightarrow \texttt{member?}(x, L_2))$

corollary: equivalence axioms for set theory

$\texttt{no-dup?}(L) \Rightarrow \texttt{set-equal?}(L, L)$

$\texttt{no-dup?}(L_1) \wedge \texttt{no-dup?}(L_2) \Rightarrow (\texttt{set-equal?}(L_1, L_2) \Rightarrow \texttt{set-equal?}(L_2, L_1))$

$\texttt{no-dup?}(L_1) \wedge \texttt{no-dup?}(L_2) \wedge \texttt{no-dup?}(L_3) \Rightarrow$
$\quad (\texttt{set-equal?}(L_1, L_2) \wedge \texttt{set-equal?}(L_2, L_3) \Rightarrow \texttt{set-equal?}(L_1, L_3))$

theorem: $\texttt{no-dup?}(L) \Rightarrow \texttt{set-equal?}(\texttt{insert}(x, \texttt{insert}(x, L)), \texttt{insert}(x, L))$
theorem: $\texttt{no-dup?}(L) \Rightarrow \texttt{set-equal?}(\texttt{insert}(y, \texttt{insert}(x, L)), \texttt{insert}(x, \texttt{insert}(y, L)))$

The set of all lists without duplications is an inductive sort generated by nil and insert:
theorem:
$\texttt{no-dup?}(L) \Leftrightarrow L = \texttt{nil} \vee \exists(x)\, \exists(L' \mid \texttt{no-dup?}(L'))\ \neg\texttt{member?}(x, L') \wedge L = \texttt{insert}(x, L')$

end-theory

2.6 Interpretations Between Theories

Interpretations (translations) [1, 2, 3] are maps from the language $\mathcal{L}_1$ of theory $\mathcal{T}_1$ into the language $\mathcal{L}_2$ of theory $\mathcal{T}_2$ that preserve the validity of the theorems of $\mathcal{T}_1$; in particular, the translations of the axioms of $\mathcal{T}_1$ must be provable from the axioms of $\mathcal{T}_2$ and the images of the sorts of $\mathcal{T}_1$ must contain the interpretations of the constants of $\mathcal{T}_1$ and must be closed under the interpretations of the functions of $\mathcal{T}_1$. Such an interpretation naturally defines models of $\mathcal{T}_1$ in models of $\mathcal{T}_2$. See Section 2.6.3.

The basis required to establish an interpretation of $\mathcal{T}_1$ in $\mathcal{T}_2$ is:

- An interpretation map I from $\mathcal{L}_1$ to $\mathcal{L}_2$
- Proofs of the translations of the axioms of $\mathcal{T}_1$ in $\mathcal{T}_2$ (including the axioms for equality, if present)

This basis must be extended to cover all definitions and theorems of $\mathcal{T}_1$. The core of this is establishing that theorems in $\mathcal{T}_1$ theorems translate to theorems in $\mathcal{T}_2$. This implies that explicit definitions in $\mathcal{T}_1$ translate to explicit definitions in $\mathcal{T}_2$.

In the following, we give examples of an interpretation map[5] and translations of axioms, using `Set-to-List` for the examples.

2.6.1 Interpretation Map

An interpretation map I is defined by first defining an auxiliary translation map I' by induction over the syntactic classes of the source language. I' in turn is defined in terms of the following four auxiliary maps:

Sort Map: I_S
For each sort s' of the source language, $I_S(s')$ is a (possibly constructed) sort of the target language. For example,
$I_S[\![\,\alpha]\!] = \beta$
$I_S[\![\,\mathtt{Set}(\alpha)]\!] = \mathtt{List}(\beta)$

Variable Map: I_V
For each variable ν of sort s' of the source language, $I_V(\nu)$ is a distinct variable of sort $I_S(s')$. For example,
Variables over α go to variables over β with the same name.
Variables over $\mathtt{Set}(\alpha)$ go to variables over $\mathtt{List}(\beta)$ with the same name.

[5] This presentation closely follows that in [3]; see it for the full details.

Relativization Map: I_R

For each sort s' of the source language, I_R is a *relativization predicate* over the sort $I_S(s')$. For example,

$I_R[\![\, \texttt{Set}(\alpha)]\!] = \texttt{no-dup?}$

$I_R[\![\, \alpha]\!] = \texttt{true?}_\beta$

(Here $\texttt{true?}_\beta$ is a predicate over the sort β which always yields the value `true`). The relativization predicate is used to restrict the target sort to just those "concrete" objects that actually represent "abstract" objects; e.g., restricting lists to non-duplicating lists. The extensions of these relativization predicates must contain the interpretations of the constants of $\mathcal{T}_1$ and be closed under the interpretations of the functions of $\mathcal{T}_1$ if the model construction in Section 2.6.3 is to succeed. In the case of interpreting $\texttt{Set}(\alpha)$ in $\texttt{List}(\beta)$ this requires (among other things) that:

$I_O[\![\, \{\,\}]\!] = \texttt{nil}$ be a list without duplicates

and that since $I_O[\![\, \texttt{with}\,]\!] = \texttt{insert}$:

if L is a list without duplicates, then $\texttt{insert}(x, L)$ is also a list without duplicates.

Operations Map: I_O

I_O maps the "operation" (relation, function, constant) symbols of the source theory to the (possibly defined) operation symbols of the target theory. For example,

$I_O[\![\, \{\,\}]\!] = \texttt{nil}$

$I_O[\![\, \texttt{with}\,]\!] = \texttt{insert}$

$I_O[\![\, \in]\!] = \texttt{member?}$

$I_O[\![\, =_{\texttt{Set}(\alpha)}]\!] = \texttt{set-equal?}$[6]

Interpretation Auxiliary Map: I'

Once the four basis maps are defined, the auxiliary map I' is defined by induction over the terms and formulas of $\mathcal{L}_1$.

Terms

Variables	$I'[\![\, \nu]\!] = I_V[\![\, \nu]\!]$
Constants	$I'[\![\, c]\!] = I_O[\![\, c]\!]$
Applications	$I'[\![\, F(\tau_1, ..., \tau_n)]\!] = I_O[\![\, F]\!]\,(I'[\![\, \tau_1]\!], ..., I'[\![\, \tau_n]\!]\,)$

Formulas

Atomic Formulas

$I'[\![\, R(\tau_1, ..., \tau_n)]\!] = I_O[\![\, R]\!]\,(I'[\![\, \tau_1]\!], ..., I'[\![\, \tau_n]\!]\,)$

[6] In a theory interpretation equality is treated as just another predicate: it can be interpreted by any equivalence relation in the target theory that is a congruence relation with respect to the interpretations of the other operations of the source theory.

Propositional Compounds

$I'[\![\neg\phi]\!] \;=\; \neg\, I'[\![\phi]\!]$
$I'[\![\phi_1 \wedge \phi_2]\!] \;= I'[\![\phi_1]\!] \;\wedge I'[\![\phi_2]\!]$
$I'[\![\phi_1 \vee \phi_2]\!] \;= I'[\![\phi_1]\!] \;\vee I'[\![\phi_2]\!]$
$I'[\![\phi_1 \Rightarrow \phi_2]\!] \;= I'[\![\phi_1]\!] \;\Rightarrow I'[\![\phi_2]\!]$

Universal Quantification

$I'[\![\forall(\nu : \sigma)\ \phi]\!]$
$\quad = \forall(I_V(\nu) : I_S(\sigma))\ I_R(\sigma)(\nu) \Rightarrow I'[\![\phi]\!]\,)$

Hence,

$I'[\![\forall(\nu : \texttt{Set}(\alpha))\ \phi]\!] \;= \forall(\nu : \texttt{List}(\beta))\ \texttt{no-dup?}(\nu) \Rightarrow I'[\![\phi]\!]$

and

$I'[\![\forall(\nu : \alpha)\phi]\!] \;= \forall(\nu : \beta)(\texttt{true}_\beta \Rightarrow I'[\![\phi]\!]\,) = \forall(\nu_\beta)\ I'[\![\phi]\!]$

Existential Quantification

$I'[\![\exists(\nu : \sigma)\ \phi]\!]$
$\quad = \exists(I_V(\nu) : I_S(\sigma))\ (I_R(\sigma)(\nu) \wedge I'[\![\phi]\!]\,)$

Hence,

$I'[\![\exists(\nu : \texttt{Set}(\alpha))\ \phi]\!] \;= \exists(\nu : \texttt{List}(\beta))\ \texttt{no-dup?}(\nu) \wedge I'[\![\phi]\!]$

and

$I'[\![\exists(\nu : \alpha)\ \phi]\!] \;= \exists(\nu : \beta)(\texttt{true}_\beta \wedge I'[\![\phi]\!]\,)$

Interpretation Map

Once I' is defined we can define the interpretation of arbitrary formulas (with possibly free variables) as follows:

Formulas

Let $\phi[x_1, ..., x_n]$ be a formula with free variables $x_1, ..., x_n$ of sorts $s_1, ..., s_n$. Then:

$$
\begin{aligned}
& I[\![\phi[x_1, ..., x_n]]\!] \\
= \\
& \quad I_R[\![s_1]\!]\,(I_V[\![x_1]\!]\,) \wedge ... \wedge I_R[\![s_n]\!]\,(I_V[\![x_n]\!]\,) \\
& \qquad \Rightarrow \\
& \quad I'[\![\ \phi[I_V[\![x_1]\!]\,, ..., I_V[\![x_m]\!]\,]\ ]\!]
\end{aligned}
$$

2.6.2 Interpretation of $\texttt{Set}(\alpha)$ in $\texttt{List}(\beta)$

The interpretation map was given via examples in the previous section; here we give some examples of the translations of the $\texttt{Set}(\alpha)$ axioms (including the equality axioms) in $\texttt{List}(\beta)$; i.e., what must be proven in $\texttt{List}(\beta)$ to establish the interpretation. In each case, the result of the translation is a corollary, definition, or theorem of the theory of lists presented in Section 2.5.

Prove: $\texttt{List}(\beta) \vdash\ I[\![\, x\ \notin \{\,\}]\!]$

$$
\begin{aligned}
& I[\![\, x\ \notin \{\,\}]\!] \\
&\quad = \neg I[\![\ x\ \in \{\,\}]\!] \\
&\quad = \neg\texttt{member?}(I[\![\, x]\!]\,, I[\![\,\{\,\}]\!]\,) \\
&\quad = \neg\texttt{member?}(x, \texttt{nil})
\end{aligned}
$$

Prove: $\texttt{List}(\beta) \vdash\ I[\![\, x \in (S \texttt{ with } y) \Leftrightarrow x =_\alpha y \vee x \in S]\!]$

$$
\begin{aligned}
& I[\![\, x \in (S \texttt{ with } y) \Leftrightarrow x =_\alpha y \vee x \in S]\!] \\
&\quad = \texttt{no-dup?}(S) \Rightarrow I[\![\, x \in (S \texttt{ with } y) \Leftrightarrow x =_\alpha y \vee x \in S]\!] \\
&\quad = \texttt{no-dup?}(S) \Rightarrow I[\![\, x \in (S \texttt{ with } y)]\!]\ \Leftrightarrow I[\![\, x =_\alpha y \vee x \in S]\!] \\
&\quad = \texttt{no-dup?}(S) \Rightarrow \texttt{member?}(x, I[\![\, S \texttt{ with } y]\!]\,) \Leftrightarrow I[\![\, x =_\alpha y]\!]\ \vee I[\![\, x \in S]\!] \\
&\quad = \texttt{no-dup?}(S) \Rightarrow \texttt{member?}(x, \texttt{insert}(y, S)) \Leftrightarrow x =_\beta y \vee \texttt{member?}(x, S)
\end{aligned}
$$

Prove: $\texttt{List}(\beta) \vdash\ I[\![\,(S \texttt{ with } x) \texttt{ with } y =_{\texttt{Set}(\alpha)} (S\ \texttt{ with } y) \texttt{ with } x]\!]$

$$
\begin{aligned}
& I[\![\,(S \texttt{ with } x) \texttt{ with } y =_{\texttt{Set}(\alpha)} (S\ \texttt{ with } y) \texttt{ with } x]\!] \\
&\quad = \texttt{no-dup?}(S) \Rightarrow \\
&\qquad I[\![\,(S \texttt{ with } x) \texttt{ with } y =_{\texttt{Set}(\alpha)} (S\ \texttt{ with } y) \texttt{ with } x]\!] \\
&\quad = \texttt{no-dup?}(S) \Rightarrow \\
&\qquad \texttt{set-equal?}(I[\![\,(S \texttt{ with } x) \texttt{ with } y]\!]\,, I[\![\,(S\ \texttt{ with } y) \texttt{ with } x]\!]\,) \\
&\quad = \texttt{no-dup?}(S) \Rightarrow \\
&\qquad \texttt{set-equal?}(\texttt{insert}(y, I[\![\, S \texttt{ with } x]\!]\,), \texttt{insert}(x, I[\![\, S\ \texttt{ with } y]\!]\,)) \\
&\quad = \texttt{no-dup?}(S) \Rightarrow \\
&\qquad \texttt{set-equal?}(\texttt{insert}(y, \texttt{insert}(x, S)), \texttt{insert}(x, \texttt{insert}(y, S)))
\end{aligned}
$$

Prove: $\texttt{List}(\beta) \vdash\ I[\![\,(S \texttt{ with } x) \texttt{ with } x =_{\texttt{Set}(\alpha)} S\ \texttt{ with } x]\!]$

$$
\begin{aligned}
&\quad = \texttt{no-dup?}(S) \Rightarrow \\
&\qquad I[\![\,(S \texttt{ with } x) \texttt{ with } x =_{\texttt{Set}(\alpha)} S\ \texttt{ with } x]\!] \\
&\quad = \texttt{no-dup?}(S) \Rightarrow \\
&\qquad \texttt{set-equal?}(I[\![\,(S \texttt{ with } x) \texttt{ with } x]\!]\,, I[\![\, S\ \texttt{ with } x]\!]\,) \\
&\quad = \texttt{no-dup?}(S) \Rightarrow \\
&\qquad \texttt{set-equal?}(\texttt{insert}(x, I[\![\, S \texttt{ with } x]\!]\,), \texttt{insert}(x, S)) \\
&\quad = \texttt{no-dup?}(S) \Rightarrow \\
&\qquad \texttt{set-equal?}(\texttt{insert}(x, \texttt{insert}(x, S)), \texttt{insert}(x, S))
\end{aligned}
$$

The equivalence axioms for Set Equality

Prove: $\texttt{List}(\beta) \vdash\ I[\![\, S = S]\!]$

$$
\quad = \texttt{no-dup?}(S) \Rightarrow \texttt{set-equal?}(S, S)
$$

Prove: $\texttt{List}(\beta) \vdash I[\![S_1 = S_2 \Rightarrow S_2 = S_1]\!]$
$= \texttt{no-dup?}(S_1) \wedge \texttt{no-dup?}(S_2) \Rightarrow (\texttt{set-equal?}(S_1, S_2) \Rightarrow \texttt{set-equal?}(S_2, S_1))$

Prove: $\texttt{List}(\beta) \vdash I[\![S_1 = S_2 \wedge S_2 = S_3 \Rightarrow S_1 = S_3]\!]$
$= \texttt{no-dup?}(S_1) \wedge \texttt{no-dup?}(S_2) \wedge \texttt{no-dup?}(S_3) \Rightarrow$
$(\texttt{set-equal?}(S_1, S_2) \wedge \texttt{set-equal?}(S_2, S_3) \Rightarrow \texttt{set-equal?}(S_1, S_3))$

The substitution axioms for Set Equality

Prove: $\texttt{List}(\beta) \vdash I[\![S_1 = S_2 \Rightarrow (x \in S_1 \Leftrightarrow x \in S_2)]\!]$
$= \texttt{no-dup?}(S_1) \wedge \texttt{no-dup?}(S_2) \Rightarrow$
$\texttt{set-equal?}(S_1, S_2) \Rightarrow ((x, S_1) \Leftrightarrow \texttt{member?}(x, S_2))$

Prove: $\texttt{List}(\beta) \vdash I[\![S_1 = S_2 \Rightarrow (S_1 \texttt{ with } x = S_2 \texttt{ with } x)]\!]$
$= \texttt{no-dup?}(S_1) \wedge \texttt{no-dup?}(S_2) \Rightarrow$
$\texttt{set-equal?}(S_1, S_2) \Rightarrow \texttt{set-equal?}(\texttt{insert}(x, S_1), \texttt{insert}(x, S_2))$

Note that since the constructors of `Set` are mapped to the "constructors" of non-duplicating lists, the translation of an instance of the induction schema for sets is an instance of the induction schema for lists.

2.6.3 Interpretation Model Construction

Our notion of an implementation of a theory is a model of the theory, and our notion of refinement is an operation mapping a theory $\mathcal{T}_1$ to a theory $\mathcal{T}_2$ such that every model of $\mathcal{T}_2$ is also a model of $\mathcal{T}_1$. I.e., refinement restricts the models that may be used as an implementation. This is an abstract, model theoretic condition. At an operational level, we use theory interpretations, a proof-theoretic concept, to specify refinements. We need to explain, given an interpretation from $\mathcal{T}_1$ to $\mathcal{T}_2$, how to construct a model of $\mathcal{T}_1$ from a model of $\mathcal{T}_2$. Note that $\mathcal{T}_2$ may be over a different language than $\mathcal{T}_1$ so models of $\mathcal{T}_2$ cannot be directly viewed as candidate models of $\mathcal{T}_1$.

The existence of an interpretation from $\mathcal{T}_1$ to $\mathcal{T}_2$ enables the use of general model-theoretic methods for constructing models $\mathcal{M}_2|I$ (*the model induced by I in $\mathcal{M}_2$*) of $\mathcal{T}_1$ from models $\mathcal{M}_2$ of $\mathcal{T}_2$. In broad outline, for theories with equality, that method[2] is the following.

Sorts–The domains representing the sorts of $\mathcal{M}_2|I$ consist of equivalence classes of elements of the domains of $\mathcal{M}_2$ induced by the equivalence relation E that is the interpretation of the equality relations in $\mathcal{T}_1$ restricted to the domain elements satisfying the relativization predicates. In our example, this is the denotation of `set-equal?` restricted to `no-dup?` together with the equality relation on β. Because the interpretations of the axioms for $=$ in $\mathcal{T}_1$ hold in $\mathcal{T}_2$ (including the equivalence and substitution axioms), E will be a congruence relation.

Equalities–The equality relation on a given sort of $\mathcal{M}_2|I$ is the equality over the equivalence classes of E restricted to that sort.

Relations–The relations of $\mathcal{M}_2|I$ are the denotations of the interpretations of the relations of $\mathcal{T}_1$ factored by E.

Functions–The functions of $\mathcal{M}_2|I$ are the denotations of the interpretations of the functions of $\mathcal{T}_1$ factored by E.

Constants–The constants of $\mathcal{M}_2|I$ are the equivalence classes of E containing the denotation of the interpretation of constant symbol.

As mentioned in the previous section, in order for the construction to actually define a first order structure for $\mathcal{L}_1$, we must show that the relativized interpretations of the sorts of $\mathcal{T}_1$ contain the interpretations of the constants of $\mathcal{T}_1$ and are closed under the interpretations of the functions of $\mathcal{T}_1$ and that E is a congruence relation.

The following theorem and its corollary are from [2].

Theorem:[7] Let I be an interpretation of $\mathcal{L}_1$ in $\mathcal{T}_2$ and let $\mathcal{M}_2$ be a model of $\mathcal{T}_2$. Let ϕ be a formula of $\mathcal{L}_1$, let γ be an assignment of values in $\mathcal{M}_2$ to the free variables of $I[\![\phi]\!]$, and let γ' be the assignment that maps a variable ν to the equivalence class of $\gamma(\nu)$. Then

$$\gamma' \models_{\mathcal{M}_2|I} \phi \Leftrightarrow \gamma \models_{\mathcal{M}_2} I[\![\phi]\!]$$

Corollary: If I is an interpretation of $\mathcal{T}_1$ in $\mathcal{T}_2$, $\mathcal{M}_2$ is a model of $\mathcal{T}_2$, and ϕ is a theorem of $\mathcal{T}_1$, then $\models_{\mathcal{M}_2|I} \phi$.
Proof:
If ψ is an axiom of $\mathcal{T}_1$, then $I[\![\psi]\!]$ is a theorem of $\mathcal{T}_2$. By the theorem, $\models_{\mathcal{M}_2|I} \psi$. Hence, $\mathcal{M}_2|I$ is a model of $\mathcal{T}_1$ and $\models_{\mathcal{M}_2|I} \phi$.

Hence, we have the following.
Theorem:[1, 2] If I is an interpretation of $\mathcal{T}_1$ in $\mathcal{T}_2$, then there is a construction (the *induced model* construction) that transforms models of $\mathcal{T}_2$ into models of $\mathcal{T}_1$.

[7]It is this theorem that is at the core of the generalization of interpretations in first order logic to institutions (see [11]).

2.6.4 Definitional Extensions

Let D be an explicit definition of a function in $\mathcal{T}_1$; say D has the form: $f(x) = y \Leftrightarrow \phi[x, y]$. In order for D to be a justified definition of $\mathcal{T}_1$, it must be the case that justification condition (the pointwise existence and uniqueness condition) is a theorem of $\mathcal{T}_1$:

$$\models_{\mathcal{T}_1} \forall(x)\, \exists!(y)\, \phi[x, y]$$

If I is an interpretation of $\mathcal{T}_1$ in $\mathcal{T}_2$, then the interpretation of the justification condition for D will be a theorem of $\mathcal{T}_2$[8]. Hence, mapping a defined symbol f of $\mathcal{T}_1$ to a defined symbol f' whose definition in $\mathcal{T}_2$ is the image under I of the definition of f in $\mathcal{T}_1$ always yields a legitimate extension of the interpretation: justification conditions will be mapped to justification conditions. It follows that establishing the correctness of the extended interpretation is simply a matter of matching formulas. Further, if the interpretation is specified by giving the interpretation of the primitives of $\mathcal{T}_1$ and generating the definitions in $\mathcal{T}_2$ that are the images of the definitions in $\mathcal{T}_1$, then the extensions are guaranteed to be correct.

For inductive definitions, the situation is similar: if the constructors are mapped to constructors, then the induction schemas will translate to induction schemas and legitimate inductive definitions will translate to legitimate inductive definitions.

2.7 Data Type Refinement via Interpretations

The core notion of refinement used here is the same as that of Sannella and Tarlecki in [7, 8]:

S **refines to** S' iff every model of S' is a model of S.

The notion of refinement utilized for data type refinement is the generalization called *constructor implementation* in [8]:

S **is implemented by** S' **via the constructor** k
iff
k transforms models of S' into models of S.

These notions are model-theoretic notions; the actual techniques employed in our methods of data type refinement are proof-theoretic: the refinement is presented as an interpretation between theories. The link between them is provided by the induced model construction of Section 2.6.3. The correctness of refinement by interpretation follows directly from the fact that the induced model construction transforms models of the target

[8] Where the "there-exists-a-unique" quantifier is interpreted to mean "unique up to E-equivalence".

theory into models of the source theory.[9] Also, the abstraction or "retrieve" functions of Hoare are given by the mapping taking elements of the model of $\mathcal{T}_2$ to the equivalence class containing the element. Hence, the refinements are correct in the sense of [17].

2.8 Algebra of Interpretations

The atomic interpretations are interpretations from atomic data types into atomic data types; e.g., the interpretation `Std-Integer` interprets the integers in COMMON LISP integers. The interpretations of parameterized theories are themselves parameterized: `Set-to-CL-List`(β) maps `Set`(α) to lists in COMMON LISP of elements whose type is α and whose implementation is β. The following is a simple example of a compound interpretation constructed via parametric composition:

$$\texttt{Set-to-CL-List}(\texttt{Set-to-CL-Bitvector}(\beta))$$

Interpretations can also be composed sequentially, which we will express using $\circ$. For example the interpretation `Set-to-CL-List`(β) can be factored into the two interpretations `Set-to-List`(β) and `List-To-CL-List`(β) and expressed as:

$$(\texttt{List-To-CL-List} \circ \texttt{Set-to-List})(\beta)$$

The basic interpretations form a category: the existence of identity interpretations is trivial and the composition operator $\circ$ is easily shown to be associative.

3 DTRE Transformation System

The Data Type Refinement Environment (DTRE) is a synthesis system that utilizes directives to synthesize efficient code from specifications. Logically its core is the use of interpretations between theories to implement data type refinements in a provably correct manner. In system terms, the core of the system is the facility for annotating specifications with annotations directing that particular objects and operations be given specific implementations together with the transformation system that refines specifications according to those directives. This separation of types and their implementations permits specification to proceed independently of implementation while simultaneously providing a basis for rapid and verifiably correct transformation to efficient code. Users can change implementations by simply changing annotations. Among other things this provides a smooth transition between prototyping, producing a production quality system, and maintaining the system. DTRE itself is implemented in REFINE[18] and has been

[9] See [16] for a recent survey of the semantics of Algebraic Specifications.

integrated with the KIDS system[4] for algorithm design and refinement. We first give a simple example of what the system does and then give brief descriptions of some parts of the system. In the example, the user has not specified any implementations; hence, the system will try to select reasonable implementations. The primary flow of control is: type checking and inference; data-flow and bounds analysis; data structure selection; and, finally, refinement.

We first give a simple example of what the system does and then give brief descriptions of some parts of the system. In the example, the user has not specified any implementations; hence, the system will try to select reasonable implementations. The primary flow of control is: type checking and inference; data-flow and bounds analysis; data structure selection; and, finally, refinement.

3.1 Example: Job Scheduling

`Schedule-Jobs` is an instance of topological sort: given a partial order it finds a linear order that respects the partial order.

The algorithm presented below is the algorithm produced by the KIDS system: the basic form of the algorithm has been determined by KIDS, the remaining task is to select efficient data structures. The algorithm is a workset algorithm: each time through the loop, the algorithm selects for scheduling some job all of whose predecessors have already been scheduled. The actual algorithm below improves upon the basic algorithm by keeping track of the number of unscheduled predecessors of a job and the immediate successors of each job. This information is used to maintain the set of jobs with no predecessors.

Type declarations for `Schedule-Jobs`

```
Type  JobsSet    =  Set(Integer)
Type  JobsReln   =  Set(Tuple(Integer,Integer))
Type  JobsSeq    =  Seq(Integer)
Type  SuccMap    =  Map(Integer,set(Integer))
Type  NumMap     =  Map(Integer,Integer)
Type  IntWorkSet =  Set(Integer)
```

Specification for `Schedule-Jobs`

This is the form of the specification as it enters DTRE.

```
Function  Schedule-Jobs
  ( S : JobsSet, R : JobsReln  | Domain(R) = S )
    returns ( p : JobsSeq | range(p) = S )
 =
  (Let  (Sz = size(S),
        result  =  [])
    (Let (Succ: SuccMap   = {| v -> {}| (v) v in S |},
          NumPred: NumMap = {| v -> 0 | (v) v in S |}
         |  Domain(Succ) = S &  Size(Succ) = Sz &
            Domain(NumPred) = S & Size(NumPred) = Sz )

     (Enumerate z over R do
        (let (x : Integer = z.1,  y : Integer = z.2 )
           Succ(x)  <-   Succ(x) with! y;
           NumPred(y)  <-   NumPred(y) + 1));

     (Let (MinEls: IntWorkSet = {})
        (Enumerate v over S do
            if NumPred(v) = 0
              then  MinEls  <-  MinEls with! v);

          (While  Sz > 0 do
            (Let (a = arb(MinEls))
               MinEls  <-  MinEls less! a;
               result  <-  append(result,a);
               (Enumerate w over Succ(a) do
                  NumPred(w)  <-  NumPred(w) - 1;
                  if NumPred(w) = 0
                   then  MinEls  <-  MinEls with! w);
               Sz  <-  Sz - 1)))  );
        result)
```

`Schedule-Jobs` **With Major Implementation Directives**

This is the form of the specification after data structure selection; the implementations were selected by the DSS algorithm. Other annotations have been omitted for readability.

```
Function  Schedule-Jobs
  ( S : JobsSet, R : JobsReln  | Domain(R) = S )
    returns ( p : JobsSeq | range(p) = S )
 =
  (Let  (Sz = size(S),
        result  =  [])
    (Let (Succ: SuccMap
            impl-by MAP-TO-ARRAY(STD-INTEGER, SET-TO-LIST(STD-INTEGER))
            = {| v -> {}| (v) v in S |},
          NumPred: NumMap
            impl-by MAP-TO-ARRAY(STD-INTEGER, STD-INTEGER)
            = {| v -> 0 | (v) v in S |}
         |  Domain(Succ) = S &  Size(Succ) = Sz &
            Domain(NumPred) = S & Size(NumPred) = Sz )

     (Enumerate z over R do
        (let (x : Integer = z.1,  y : Integer = z.2 )
           Succ(x)  <-   Succ(x) with! y;
           NumPred(y)  <-   NumPred(y) + 1));

     (Let (MinEls: IntWorkSet impl-by SET-TO-STACK(STD-INTEGER) = {})
        (Enumerate v over S do
            if NumPred(v) = 0
              then  MinEls  <-  MinEls with! v);

          (While  Sz > 0 do
            (Let (a = arb(MinEls))
               MinEls  <-  MinEls less! a;
               result  <-  append(result,a);
               (Enumerate w over Succ(a) do
                  NumPred(w)  <-  NumPred(w) - 1;
                  if NumPred(w) = 0
                   then  MinEls  <-  MinEls with! w);
               Sz  <-  Sz - 1)))  );
        result)
```

Common Lisp **Code Generated For** `Schedule-jobs`

```
(DEFUN SCHEDULE-JOBS (S R)
  (LET ((SZ (LENGTH S)) (RESULT (LIST)))
    (LET ((SUCC (LET ((MAPVAR-1
                        (CREATE-CONSTANT-ARRAY1 SZ (QUOTE *UNDEFINED*))))
                  (LOOP FOR V IN S DO (SETF (SVREF MAPVAR-1 V) NIL))
                  MAPVAR-1))
           (NUMPRED
             (LET ((MAPVAR-0
                     (CREATE-CONSTANT-ARRAY1 SZ (QUOTE *UNDEFINED*))))
               (LOOP FOR V IN S DO (SETF (SVREF MAPVAR-0 V) 0))
               MAPVAR-0)))
      (LOOP FOR Z IN R DO
        (LET ((X (CAR Z))
              (Y (CDR Z)))
          (SETF (SVREF SUCC X)(CONS Y (SVREF SUCC X)))
          (SETF (SVREF NUMPRED Y)(+ (SVREF NUMPRED Y) 1))))
      (LET ((MINELS NIL))
        (LOOP FOR V IN S DO
          (IF! (LET ((VALUE-0 NIL))
                 (IF! (ZEROP (SVREF NUMPRED V))
                   THEN (SETQ VALUE-0 T))
                 VALUE-0)
            THEN (SETQ MINELS (CONS V MINELS))))
        (LOOP WHILE (> SZ 0)
          DO (LET ((A (IF (NULL MINELS) (QUOTE *UNDEFINED*) (CAR MINELS))))
               (SETQ MINELS (CDR MINELS))
               (SETQ RESULT (CONS A RESULT))
               (LOOP FOR W IN (SVREF SUCC A)
                 DO (PROGN
                      (SETF (SVREF NUMPRED W)(- (SVREF NUMPRED W) 1))
                      (IF! (LET ((VALUE-1 NIL))
                             (IF! (ZEROP (SVREF NUMPRED W))
                               THEN (SETQ VALUE-1 T))
                             VALUE-1)
                        THEN (SETQ MINELS (CONS W MINELS)))))
               (SETQ SZ (- SZ 1))))))
    (NREVERSE RESULT)))
```

The Data Structure Selection (DSS) algorithm in Dtre has selected array implementations for the maps `SuccMap` and `NumPred` and has selected a stack implementation for the set `IntWorkSet`. These implementations represent a significant improvement over the default implementations of association lists for maps and lists without duplicates for sets. If the default implementations are used, the algorithm is cubic and takes approximately 18 seconds of cpu on a Sun-4 for input of size roughly 500. However, if arrays and a stack are used, the algorithm is linear and the time for processing an input of size 500 is about 15 milli-seconds of cpu—a speedup of approximately 1200 times.

3.2 System Components And Technology

There are many capabilities required of transformation systems such as DTRE; we give brief descriptions of:

- Refinements and the DTRE Language
- Data Structure Selection
- Transformation Procedure

3.2.1 Refinements and the DTRE Language

The core technology is the formulation of data type refinements as interpretations. The algebra of interpretations provides the basis for the description of compound implementations as compositions of basic interpretations; and, hence, provides the basis of the DTRE language for specifying implementations. The DTRE language is a primary medium for communication between the user and the system as well as between parts of the system such as the DSS and the transformation system. The system provides special grammars for describing implementations, maintains a library of data type refinements, and provides both textual and menu-based methods for specifying refinements. These are described below along with the special role that equality plays in the formulation and refinement of the abstract generic types.

Descriptions of Interpretations in DTRE Logically, interpretations are meta-level maps from the language of one theory to another. Operationally, interpretations in DTRE are implemented as lists of transformation rules in the language REFINE [18] which provides an object based representation of the abstract syntax and a pattern based transformation system for manipulating the abstract syntax. We have added an *interpretation compiler* that takes a specification of the interpretation in a language close to the object level of the theories involved (with a few simple pattern notations), lifts it to the meta-level, and then utilizes a knowledge base to generate the REFINE rules including the auxiliary code to bind and test attributes on the left hand sides of rules and set them on the right hand sides of rules.

Library of Data Type Refinements There is a library of Data Type Refinements formally expressed as compositions over the basic proof theoretic interpretations. Here is a sample of some of the current basic interpretations.

Atomic Types

Each atomic type has a standard impl in the target language.
For example, `Std-Integer`, `Std-Char`, etc.

Sets

`Set-to-List`
`Set-to-BitVector`
`Set-to-iBitVector`
`Set-to-Stack`

Seqs

`Seq-to-List`
`Seq-to-Array`
`Std-String-Seq`
`Seq-to-BitVector`

Tuples

`Tuple-to-Pair`
`Tuple-to-List`

Relations

`Relation-to-Map`

Maps (Functions)

`Map-to-aList`
`Map-to-Array`
`Map-to-Code`
`Map-to-Lambda`
`Map-to-Cached-Code`

Methods for Annotating Specifications The current DTRE language for specifying implementations is essentially the language of the algebra of interpretations: terms are constructed from parameterized interpretations via both parametric and sequential composition. As mentioned in the previous sections, users can annotate program objects with these terms as implementation directives. This is supported textually: programs with such annotations can be parsed in and out. The following is a simple example of the use of a compound implementation directive in DTRE to annotate the declaration of a program variable V.

V : `Set(Set(Integer)) impl-by Set-to-CL-List(Set-to-CL-Bitvector(Std-Integer))`

There is also mouse driven menu support for annotating program objects. We have prototyped a performance tuning environment based on DTRE[10] that relies solely on mouse driven menu selections for annotating program objects. In that environment specifications are tuned to particular classes of data via a loop of **Select**, **Test**, and **Analyze**. Here **Select** means to select implementations. The selection is *semi-automatic*: the user specifies some implementations (possibly none) and the program selects the rest (see below). **Test** means to run the synthesized program over the test data. **Analyze** means timing analysis and profiling to spot performance bottlenecks. In this performance tuning environment, all implementation selections by the user are via mouse clicks on menus of applicable implementations.

Generics and Equality Some of the abstract data types are generic (e.g., $\mathtt{Seq}(\alpha)$) in that they take other data types (with equality) as parameters. In such data types, notions are defined in terms of the equality of the parameter type; for example, x is a member of the sequence S can be defined by

$$\exists(i \mid 1 \leq i \leq \mathtt{size}(S))\, S(i) = x$$

where the $=$ is equality on the parameter type. The function that implements the $=$ is bound to it at compile time. Usually the value is the equality of an atomic data type or a compound data type all of whose values have the same implementation. But occasionally the equality is between two objects of the same type but different implementations. In such cases, the system can utilize the proceduralization facility described below to generate an equality procedure with the appropriate implementations; this can generate (recursively) a collection of such auxiliary procedures. This technique provides the basis for the option of finessing certain conversions. An equality between sets of differing implementations could be implemented by first converting and then applying a "pure" equality procedure or it could be implemented as an equality between sets of differing implementations; in effect pushing the conversion inside. This often permits the conversion to be finessed: for example, if S_1 is a set of integers implemented as a list and S_2 is a set of integers implemented as a bitvector, then the set equality procedure will be implemented as a procedure that first compares the sizes and then iterates over S_1 testing the elements of S_1 for membership in S_2. At no point is any object converted from one representation to another.

If the implementing language supports it (as, for example, COMMON LISP does), the system can translate generic procedures into procedures where the generic function parameters are passed as arguments to the implementing function. For example, the generic abstract data type $\mathtt{Set}(\alpha)$ is translated by `Set-to-CL-List` into COMMON LISP procedures with extra function arguments for the equality of the parameter type. The membership function on $\mathtt{Set}(\alpha)$ is translated to

$$(\mathtt{member}\ x\ S\ \mathtt{:test\ eqfun})$$

The generated rules that transform membership tests contain code to look up or create the appropriate equality function and substitute its name for `eqfun` in the call.

3.2.2 Data Structure Selection

The term *Data Structure Selection* (DSS), refers to selecting implementations for abstract objects in terms of lower level data structures such as lists, arrays, bitvectors, etc. We believe that fully automatic selection of *efficient* implementations is beyond current technology and the system we are developing is a *semi-automatic* system for data structure selection and refinement. In such a semi-automatic system the user supplies some implementation directives and the system determines the rest and refines the specification to code. The DSS algorithm has three phases:

Propagate During this phase user supplied and readily available information is propagated and collected at appropriate program points. Some examples are: the operations on a given program object, user assertions about sizes, etc.

Analyze Here the program is analyzed to determine such properties as data flow, value flow, size bounds, base bounds, etc. needed to make reasonable data structure selections.

Select

- First, "plausible" DSS rules are use to filter possible implementations and arrive at a set of plausible implementations.
- Second, simple heuristics are utilized to select from among the plausible implementations. (This is being upgraded[10] to select from the collection of plausible implementations on the basis of a qualitative measure of the "cost" of the operations on a given implementation weighted by the "importance" (frequency) of the operations.)

3.2.3 Transformation Procedure

The overall transformation procedure is an exhaustive top down application of the transformation rules to the abstract syntax tree of the specification or program. An important auxiliary facility is the *Proceduralization Facility* which is used at compile time to construct new procedures and translation rules according to specifications generated by other transformation rules.

Structure of the Transformation Map The rules are organized into an indexed map with the primary index being the class of the node (of the abstract syntax tree). The rules for each node class are placed into three groups:

Simplification and Normalization Rules—Used to reduce the kinds of syntactic forms seen by the main translation rules.

Translation Rules—These are the main interpretation rules. They are organized into a tree indexed by the dominant terms of the implementation specification for the current node. An attempt is made to order them most specific first. The indexing is good enough that it is rare to have more than one applicable rule.

Backstop Rules—These are rules that apply when a node should be translated but there is no specific translation rule that applies. The usual case is that it is an operation with mixed implementations; e.g., $A \cup B$ where the implementations of A and B differ. In such cases the system passes a general definition for $\cup$ (i.e., at the level of set theory rather than at the level of an implementation) to the proceduralization facility which then uses the actual implementations attached to the node (and its subnodes) to construct a $\cup$ procedure in the target language. It then constructs a translation rule to recognize similar instances of $\cup$ and to translate them to calls on the generated procedure. It also inserts the new translation rule into the transformation map.

Proceduralization Facility and Data Base The proceduralization facility provides two main services:

Procedure Generation Take an appropriate piece of the program tree, instructions about which pieces correspond to formal parameters, and implementation directives for the program and generate a procedure. (This is essentially lambda abstraction with implementation directives.)

Rule Generation Generate a translation rule to recognize instances of the generated procedure body and replace them by a call to the generated procedure.

The system handles either generic or ground structure. It also maintains a data base of such procedures and rules.

4 Related Work

This paper describes a system for data refinement and its theoretical foundations. The system itself is well developed and has been used to compile a language with set-theoretic data types into Common Lisp. Much of the theoretical framework reflects requirements derived from experience with the system.

Our formalism for specification of abstract types is based on standard notions from first-order logic (theory, conservative extension, models, and interpretations), in contrast to a more algebraic approach derived from restricting axioms to universally-quantified conditional equations. This, in general, excludes term-rewriting as a generic implementation technique [6, 19], but it is our thesis that pervasive use of axiomatically defined types is dependent on the ability to generate highly efficient representations for these types.

In this paper only the most basic facilities for defining types are presented. Hierarchical structuring and partial functions as in Prospectra [20], parameterization as in OBJ and RAISE [6, 21, 22], and non-determinism are features we regard as essential. Note that the manner in which DTRE treats polymorphism generalizes to the richer parameterization of OBJ3.

There is a considerable body of literature on the theoretical foundation for data refinement; we only supply a few representative references. Hoare [17] was among the first to address the problem and introduced the use of abstraction functions to state correctness conditions for the implementation of an "abstract" type in terms of a "concrete" type. Burstall and Darlington demonstrated a transformational approach to data refinement in [23].

An extensive amount of work has been developed in the context of algebraically-specified data types. Here the notions of signature, signature morphism, theory morphism, etc. were introduced into the Computer Science community along with the use of theories for specifications. See, for example, [24, 25, 26, 19]. Also, much of this work has been generalized to *institutions* and given a category-theoretic formulation. See [27, 28, 24, 29, 11]. Also, see [16] for a recent survey of Algebraic Specifications.

Much of the current work on refinement is done in the context of the Dijkstra-Gries formalism using predicate transformers as the semantic basis [30, 31, 32, 33]. Prins' Thesis [9] describes the design of a data refinement system based on weakest pre-condition semantics that supports implementation directives, partial implementations, and generic implementations. His goals are in agreement with ours: to formally describe reusable refinements of abstract types suitable for practical programming systems.

The refinement framework we have adopted is based on the general notions described in [7, 8]. In this formulation specifications are theories and a refinement step simply restricts the class of models of the specification; for example, by strengthening the theory. We have used theory interpretations [1, 2] to specify implementation refinements in a manner similar to that of Maibaum and Turski [34, 3]. We are also using the notion of theory interpretation to extend the generic types presented here to the more general mechanism of parameterized theories as presented in [6]. We have found that specifying data refinements with theory interpretations rather than abstraction functions as used in VDM [35] provides added generality, and, more significantly, may be directly compiled into transformation rules.

Although our methods are applicable to the refinement of an arbitrary abstract data types, we have concentrated our efforts on developing a library of refinements which can generate the large variety of implementations for set-theoretic types. Since these types are fundamental to VDM and Z [36], our work will support the formal machine-mediated refinement of VDM and Z specifications.

The SETL project [37, 38] pioneered optimization methods within a traditional compiler framework for automatic representation of set-theoretic data structures. Recent work on SETL emphasizes interactive systems for data structure selection [39]. Taliere, the system developed at NYU for interactive data structure selection utilizes user estimates of branching frequencies and size estimates to select *based* representations for sets and maps.

To our knowledge there is no work in which as comprehensive a system as DTRE for data refinement is actually implemented. Our work also derives from early experimental systems as described in Barstow [40] and Kant [41], Kotik and Goldberg [42]. The earliest designs based on using theory interpretations for refinements and terms over the extended algebra of interpretations for implementation directives were done in 1986.

5 Bibliography

References

[1] Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.

[2] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[3] Wladyslaw M. Turski and Thomas E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, Wokingham, England, 1987.

[4] Douglas R. Smith. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering*, 16(9):1024–1043, September 1990.

[5] Michael A. Jackson. *System Development. International Series in Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[6] Joseph A. Goguen and Timothy Winkler. *Introducing OBJ3.* Technical Report SRI-CSL-88-09, SRI International, Menlo Park, California, 1988.

[7] Donald Sannella and Andrzej Tarlecki. Program specification and development in Standard ML. In *Twelfth ACM Symposium on Principles of Programming Languages*, pages 67–77, New Orleans, LA, January 14–16, 1985.

[8] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25(3):233–281, 1988.

[9] Jan F. Prins. *Partial Implementation in Program Derivation.* Technical Report 87-854, Cornell University, Department of Computer Science, August 1987.

[10] Lee Blaine. *Semi-Automatic Data Structure Selection.* Technical Report, Kestrel Institute, August 1990. Kestrel Institute Internal Report.

[11] Joseph A. Goguen and Rod M. Burstall. *Institutions: Abstract Model Theory for Computer Science.* Technical Report CSLI-85-30, Stanford University, 1985.

[12] Zohar Manna and Richard Waldinger. *Logical Basis for Computer Programming, Vol. 1: Deductive Reasoning.* Addison-Wesley, Reading, MA, 1985.

[13] Zohar Manna and Richard Waldinger. *Logical Basis for Computer Programming, Vol. 2: Deductive Systems.* Addison-Wesley, Reading, MA, 1990.

[14] Patrick Suppes. *Introduction to Logic.* Van Nostrand, Princeton, N.J., 1957.

[15] Z. Manna. *Mathematical Theory of Computation.* New York, McGraw-Hill, 1974.

[16] Yellamraju V. Srinivas. *Algebraic Specification: Syntax, Semantics, Structure.* Technical Report 90-15, University of California, Irvine, June 1990.

[17] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[18] *The* $REFINE^{TM}$ *User's Guide.* 1985.

[19] J. V. Guttag, J. J. Horning, and A. Modet. *Report on the Larch Shared Language: Version 2.3.* Technical Report 58, DEC Systems Research Center, April 1990.

[20] Manfred Broy. *The PROSPECTRA Methodology.* Technical Report, Universität Passau, November 1986.

[21] Erik Meiling and Chris W. George. *The RAISE Languages and Method.* Technical Report 315, ESPRIT, Dansk Datamatik Center, September 1986.

[22] Søren Prehn. From VDM to RAISE. In D. Bjørner and C. B. Jones, editors, *VDM '87: VDM — A Formal Method at Work*, pages 141–149, Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science, Vol. 252.

[23] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[24] R. M. Burstall and J. A. Goguen. Algebras, theories, and freeness: an introduction for computer scientists. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology (Marktoberdorf NATO Summer School)*, pages 329–349, D. Reidel, 1982.

[25] J. W. Thatcher, E.G. Wagner, and J.B. Wright. Data type specification: parameterization and the power of specification techniques. *IEEE Transactions on Software Engineering*, 4(4):711–732, October 1982.

[26] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, IJCAI, Cambridge, MA, August 22–25, 1977.

[27] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[28] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In A. Nielson and E. Schmidt, editors, *ICALP '82*, pages 265–281, Springer-Verlag, Berlin, 1982. Lecture Notes in Computer Science, Vol. 140.

[29] J. Meseguer and J. A. Goguen. Initiality, induction and compatibility. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541, Cambridge University Press, Cambridge, 1983.

[30] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[31] Carroll Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, 1987/88.

[32] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[33] J. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[34] T. S. E. Maibaum, Paulo A. Veloso, and M. R. Sadler. A theory of abstract data types for program development: bridging the gap. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Formal Methods and Software Development (TAPSOFT '85), Volume 2*, pages 214–230, Springer-Verlag, Berlin, 1985. Lecture Notes in Computer Science, Vol. 186.

[35] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[36] J. M. Spivey. *Understanding Z*. Cambridge University Press, Cambridge, 1988.

[37] Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26–45, January 1983.

[38] Edward Schonberg, Jacob Schwartz, and M. Sharir. An automatic technique for the selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, April 1981.

[39] Robert Straub. *Taliere: An Interactive System for Data Structuring SETL Programs.* PhD thesis, Dept. of Computer Science, New York University, May 1988.

[40] David Barstow. *Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules.* PhD thesis, Stanford University, 1977.

[41] Elaine Kant. *Efficiency in Program Synthesis.* UMI Research Press, Ann Arbor, MI, 1981.

[42] Allen Goldberg and Gordon Kotik. Knowledge-based programming: an overview of data and control structure refinement. In H. Hausen, editor, *Software Validation: Inspection, Testing, Verification, Alternatives*, pages 287–309, Elsevier, 1984. Technical Report KES.U.83.7, Kestrel Institute, October 1983.