Understanding Data Refinement Using Procedural Refinement

Steven M. Kearns
Computer Science 450
Columbia University
New York, NY 10027

Revised July 26, 1990

Columbia University Technical Report
CUCS-036-90

Abstract:

Data refinement is converting a program that uses one set of variables to an equally correct program that uses another set of variables, usually of different types. There have been a number of seemingly different mathematical definitions of data refinement. We present a unifying view of data refinement as a special case of procedural refinement, which is simpler to understand. All the data refinement theories in the literature are shown to be instances of two formulas, but we show that there are actually an infinite number of theories. In addition, we introduce the concepts of nonlocal data refinement, data refinement using semi-invariants, and procedural optimization using data refinement.

Table Of Contents

1. Introduction

Data refinement is converting a program that uses one set of variables into an equivalent program that uses a different set of variables. It is normally used to convert a high level program using high-level data types into an equivalent program using “simpler” or more machine-oriented data types. We will show that it can also be used for making “procedural” optimizations. Theories of data refinement describe in detail when a data refined program is equivalent to the original program.

Data refinement is intimately connected with program verification, which is proving that a program matches its specification. If a programmer starts with a correct program and data refines it, he must ensure that the resulting program is still correct. A theory of data refinement tells the programmer exactly what he must prove to ensure this.

Most of the complexity of a typical program arises from rearranging and maintaining various data structures. Data refinement lets one specify a program at a high level of abstraction, namely with high-level data types. Then, the program can be refined to use more efficient and machine oriented data types in a stepwise manner. [Berry et al., 76] points out that most of the problems in reasoning about pointers disappear when a program is first created with high-level abstract data types and then refined into data structures involving pointers.

Data refinement was first formalized in [Hoare, 72]. Later [Jones, 80] introduced the concepts of adequate and fully abstract representations, providing more flexible rules for data refinement. Recent interest in the mathematics of programs [Hoare and Hayes, 1987a], [Back, 88], [Morris, 87a], [Morris, 87b], [Hehner, 84], [Hehner et al., 86], has rekindled interest in data refinement. A number of more general data refinement relations have been presented recently, in a variety of formalisms.

However, some authors present their theories as a “miraculous insight”, and justify their formulas by showing that procedural refinement is a special case of data refinement. Instead, following the approach in [Hoare et al., 87b], we show that data refinement is a special case of procedural refinement, and all the relevant theories can be derived easily. By viewing the theories as arising from procedural refinement, this paper presents a clear understanding and comparison of them.

Much work dealing with parameterization of data types and refinement between data types has been done using an algebraic approach. See [Goguen et al., 78], [Guttag et al., 78], and [Kamin, 83] for an introduction. This paper concentrates instead on converting a specific program using an abstract type to a new program using a concrete type. The algebraic research concentrates on proving relations between two types T1 and T2, such that ANY program using the abstract type T1 can be translated to a program using concrete type T2 just by replacing all abstract operations on T1 by corresponding concrete operations on T2. The “specific” approach seems more general: if a specific program only cares that an integer b is odd or even, we can replace it with a boolean variable k. In general a boolean value cannot simulate an integer, but in specific programs it can. Nipkow showed that a definition similar to ↓REFINEMENT (introduced below), with an additional concept of observable values, manages to capture the essence of an abstract type “always replaceable” by a concrete type [Nipkow, 86]. His work can be seen as a link between the two approaches.

1.1. An Example of Data Refinement

The following is a simple example of data refinement. The first program prints out a list of numbers in the reverse order they are read in, using a stack.

```
program reverse
      begin
      i, item : integer;
      s : stack;

      new(s);
      for i := 1 to 10 do
            begin
            read (item);
            push (item, s);
            end;
      for i := 10 downto 1 do
            begin
            print (pop(s));
            end;
      end;
```

We have used a Pascal-like syntax here. Notice that we have used type stack as if it were a built in data type of the language. Since our language does not actually contain such a data type, we have to **data refine** the program, replacing the stack variable with other variables. The resulting program might look like this:

```
program reverse
      begin
      i : integer;
      s1 : array[1..10] of integer;

      for i= 1 to 10 do
            read (s1[i]);
      for i = 10 downto 1 do
            print (s1[i]);
      end;
```

We have replaced the variable s of type stack with variable s1 of type array. An invariant relation I holds between s and s1. Namely, s corresponds to the elements of s1, from 1 to i inclusive, thought of as a stack with a top element at s1[i]. In logic we might state this as follows:

$$I \equiv (s = \text{stack\_from\_array}(s1,\ 1,\ i))$$

The programs are equivalent in that each prints out a list of items in the reverse order they are read in. We say the second program data refines the first under invariant I.

If we had used the algebraic approach to data refinement, we would have ended up with a different program:

```
program reverse
      begin
      i : integer;
      s1 : array[1..10] of integer;
      s1_stacktop : integer;

      s1_stacktop := 1;
      for i= 1 to 10 do
            begin
            read (s1[s1_stacktop]);
            s1_stacktop := s1_stacktop + 1;
            end
      for i = 10 downto 1 do
            begin
            s1_stacktop := s1_stacktop - 1;
            print (s1[s1_stacktop]);
            end
      end;
```

A stack (with a guaranteed maximum length) can always be replaced by an array (to hold the values) and an integer (to point to the current top of the stack). In our specific program we notice that variable i can serve double duty as a loop counter and as the index of the current top of the stack. As a result, the specific notion of data refinement results in a more efficient program than the algebraic notion of data refinement.

The treatment of **partial** refinements is also more natural using "specific" refinement instead of algebraic. A partial refinement is one that is not always applicable. For example, a fixed length array can only replace a stack if we can predict the maximum depth of the stack. Another example is integers. We reason about integers as if they can be arbitrarily large. Most languages, however, only implement **bounded integers**, which have a maximum value. For a program to be correctly implemented using bounded integers, the programmer must ensure that no integer grows larger than the maximum value. Most algebraic refinements are partial because of problems like this.

2. Formalisms

Theories of data refinement are formal descriptions of when two syntactically different programs are semantically equivalent. Therefore we must describe the basics of the theory of programs as mathematical objects. A variety of formalisms have been invented for this purpose, but we concentrate on weakest preconditions.

2.1. Logic

First order predicate logic is to Computer Science what elementary algebra is to Mathematics. When dealing with the intricacies of data refinement the exacting formalism of logic is welcome. In this paper we relegate almost all proofs to section 8, so that the casual reader can understand the presentation without wading through proofs. This section describes the notation we use in this paper.

Logic formulas are composed of the operators = (equality); $\Rightarrow$ (implication); $\wedge$ (and); $\vee$ (or); $\neg$ (not); $\forall$ (universal quantification); $\exists$ (existential quantification); variables (e.g. b, k, o); predicates (e.g. P, Q, X); and functions. Also used are $\equiv$ and $\Leftrightarrow$ for implication in both directions. = binds tightest, followed by $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$, $\equiv$, $\Leftrightarrow$. We try to parenthesize whenever it makes things clearer. A variable is said to occur free in a formula if it is not in the scope of a universal or existential quantification of the variable. If a variable is not free then it is bound. For example,

$$(\forall b{:}\exists c{:}P \wedge \neg Q \vee b{=}b\emptyset \wedge \neg P \Rightarrow (\exists d{:}\neg d{=}g \Rightarrow Q \wedge P))$$

has free variables bø and g, bound variables b, c, and d, and it should be parsed as follows:

$$\forall b{:}\exists c{:}(((P \wedge (\neg Q)) \vee ((b{=}b\emptyset) \wedge \neg P)) \Rightarrow \exists d{:}(\neg(d{=}g) \Rightarrow (Q \wedge P)))$$

Most formulas will be surrounded by [brackets] which indicate universal quantification over all the variables. However, an expression modified by brackets, such as X[e/b, f/c], is X with e substituted for all free occurrence of b, and then f substituted for all free c (the order is sometimes important). When substituting into an expression, it may be necessary to change the names of some of the bound variables. For example,

$(\forall b\colon b>0 \Rightarrow (b<c \ \wedge \ c>d))[b+5/c]$ is actually equal to

$(\forall b1\colon b1>0 \Rightarrow (b1<b+5 \ \wedge \ b+5>d))$.

We rename the b's in the original formula to b1's to avoid interaction with the b+5 for c substitution.

At the beginning of section 8 we list most of the predicate calculus rules used in the paper. Proofs are sequences of formula separated by = or ⇔ (meaning the previous formula is equal to the next), ⇒ (meaning the previous formula implies the next), ⇐ (meaning the previous formula is implied by the next), or ≤ (meaning the previous program refines to the next). Each separator may have a comment, { in braces }, justifying the step.

2.2. Weakest Preconditions and Strongest Postconditions

In general, programs are thought of as taking states to states. The states are tuples of values for variables, one value for each variable. In a program with integer variables i,j, and k, one state might be <i=1, j=-3, k=18>. Predicates correspond to sets of states.

Dijkstra invented the weakest-precondition methodology in order to be able to formalize refinement [Dijkstra, 76]. The weakest precondition, written wp(s, X) or s.X, is the weakest (most general) predicate such that executing statement s will terminate, and will terminate in a state satisfying X. s is sometimes called a predicate transformer, because it translates a predicate X to a new predicate s.X. As an example of a weakest precondition consider the nondeterministic if..fi statement, which looks like this:

```
if
      | P1 → s1
      | P2 → s2
fi
```

To save space we normally write the statement as "if | P1 → s1 | P2 → s2 | fi". Intuitively, it acts by executing statement s1 if predicate P1 is true or s2 if P2 is true. If P1 and P2 are both true it nondeterministically executes s1 or s2. If neither P1 or P2 are true it is an error, and the program aborts. The weakest precondition of if...fi is shorter to describe:

$$\begin{aligned}&(\text{if} \mid P1 \rightarrow s1 \mid P2 \rightarrow s2 \text{ fi}).X \equiv \\ &\qquad (P1 \vee P2) \wedge (P1 \Rightarrow s1.X) \wedge (P2 \Rightarrow s2.X)\end{aligned}$$

To understand the rule, remember that it should specify the most general condition for the if..fi statement to terminate in a state satisfying X. The $(P1 \vee P2)$ clause ensures that the if..fi statement does not abort; the other two clauses ensure that if either branch is taken then X will be established.

There is a weakest precondition rule for each statement in the programming language we will describe next. They should be looked at as **defining** the meaning of the statement. Weakest preconditions convert a program into a logic statement. Here are definitions of several statements that make up a simple programming language, from [Dijkstra, 76] and [Morris, 87b]. In general, we use P, Q, X for predicates and s, t for program statements.

Definition 2.2.1: Weakest Preconditions

(skip).X ≡ X
(b := e).X ≡ X[e/b], where X[e/b] means X with all free occurrences of b replaced by expression e.
(b : P).X ≡ (∀b:P ⇒ X), where b may be a list of variables
{P}.X ≡ P ∧ X
{b?Q}.X ≡ (∀b:Q ⇒ X) ∧ Q, where b may be a list of variables
{?Q}.X ≡ [Q ⇒ X] ∧ Q
(p;q).X ≡ p.(q.X)
|[Tx x; t]|.X ≡ (∀x:x in Tx: t.X)
|[label$foo; t]|.X ≡ (t.X)[b/b$foo] for b a list of all variables
(if | P_i → t_i).X ≡ (∃i:P_i) ∧ (∀i:P_i ⇒ t_i.X)
(do P → t od).X ≡ (μY:(P ⇒ t.Y) ∧ (¬P ⇒ X))

“skip” is a statement which does nothing. The prescription statement, “x:P”, was introduced in [Morris, 87a] and [Morgan and Robinson, 87a]. It means “assign to variable(s) x to make predicate P true.” It cannot usually be directly implemented, but it is an important specification statement which is eventually replaced by a sequence of simpler statements.

{P} is an assertion statement; it says that at this point in the program predicate P is true. An assertion is like a comment, in that it can be deleted from a program without changing the program's behavior. However, an assertion may be used to carry out a future refinement. {b?Q} is a reassertion statement, invented by the author; it says that at this point in the program the only thing we care about variable b is that its value makes Q true. {?Q} says that the only purpose of the preceding code is to establish Q. Actually, reassertions are used to forget some of what we know at a certain point in the program. We will often use {b?TRUE}, which means at this point b can be anything. At the end of the block declaring b, for example, one can insert {b?TRUE} because its value before it disappears does not matter. Reassertions, like assertions, can safely be deleted from a program.

|[TypeB b; t]| is a local block defining variable b of type TypeB. Statement t may use b. For simplicity we assume that the name b does not conflict with any names in the enclosing blocks. |[label$foo; t]| declares a constant called b$foo for each program variable b and sets b$foo to the value of b. This allows statement t to refer to initial or intermediate values. For simplicity we assume that the label is chosen so that no generated name conflicts with any names in the enclosing blocks.

p;q is statement p followed by statement q. (Note that p and q are arbitrary programs.) Finally, do...od is a looping construct that loops while P is true. Its weakest precondition uses the least fixpoint operator μ. (Since we will not need the least fixpoint later, we do not introduce it further.) We have omitted recursion and procedures for simplicity of presentation.

By applying the above rules recursively, one can calculate the weakest precondition of **any** program with respect to a specific X. In general this formula will be long and complicated, though. As an example we will calculate the wp() of the following program s1, which is a very bad program for setting c to the square root of b.

```
s1:   if
      | (c = b) →         {c=b} c := 4*c;
      | (c <> b) →        b := c*c - 3;
                          c := c + 3;
      fi
```

Our X will be (c*c = b). As befits our bad program, X is a bad predicate: it neglects to specify that we should only change variable c during the program. In any case, applying the previous formulas we get the following:

wp(s1,x)

$\Leftrightarrow (c=b \vee c\neq b) \wedge (c=b \Rightarrow (c=b \wedge X[4c/c] \wedge (c\neq b \Rightarrow X[(c+3)/c, (c^2-3)/b])$

$\Leftrightarrow \text{True} \wedge (c=b \Rightarrow 16c^2=b) \wedge (c\neq b \Rightarrow (c+3)^2=(c^2-3))$

$\Leftrightarrow (c=b \Rightarrow 16c^2=b) \wedge (c\neq b \Rightarrow c=-2)$

$\Leftrightarrow (c=b \Rightarrow (c=b=0 \vee c=b=1/16)) \wedge (c\neq b \Rightarrow c=-2)$

$\Leftrightarrow c=b=0 \vee c=b=1/16 \vee (c=-2 \wedge b\neq -2)$

In a purely mechanical manner we have calculated all the starting conditions that ensure s1 will terminate with (c*c = b). We can now see why s1 is a terrible program for setting c to the square root of b. It only behaves as desired under the special conditions calculated above.

In order to provide simple formulas for the theories of data refinement, we also have to introduce strongest postconditions, sp(s, X), which is the strongest predicate one can assume if s terminates after starting in a state satisfying X. Here are rules for each type of statement.

Definition 2.2.2: Strongest Postconditions

$sp(skip, X) \equiv X$

$sp(b := e, X) \equiv (\exists d{:}X[d/b] \wedge b=e[d/b])$

$sp(b{:}P, X) \equiv (\exists b{:}X) \wedge P$ (b may be a list of variables)

$sp(\{P\}, X) \equiv P \wedge X$

$sp(\{b?Q\}, X) \equiv (\exists b{:}Q \wedge X) \wedge Q$

$sp(\{?Q\}, X) \equiv (\exists b{:}Q \wedge X) \wedge Q$ for b a list of all variables

$sp(p;q, X) \equiv sp(q, sp(p, X))$

$sp(|[\ Tx\ x;\ t\ ]|, X) \equiv (\exists x{:}x \text{ in } Tx{:}\ sp(t, X))$

$sp(|[\ label\$foo;\ t\ ]|, X) \equiv (\exists b\$foo{:}\ sp(t, X \wedge b\$foo=b))$ for b a list of all vars

$sp((if \mid P_i \rightarrow t_i), X) \equiv (\exists i{:}\ sp(t_i, P_i \wedge X))$

$sp((do\ P \rightarrow t\ od), X) \equiv \neg P \wedge (\mu Y{:}X \vee sp(t, P \wedge Y))$

Returning to our strange program s1, we will calculate sp(s1, W), where the initial condition W is $(c=b=0 \vee (c=-2 \wedge b\neq-2))$. (W contains two of the conditions that ensure that s1 terminates with c*c=b.) Plugging into the formulas, we derive the following:

sp(s1, W)

$\Leftrightarrow sp(\{c=b\}; c:=4*c, (c=b=0)) \vee sp(b:=c*c-3; c:=c+3, (c=-2 \wedge b\neq-2))$

$\Leftrightarrow sp(c:=4*c, (c=b=0)) \vee sp(c:=c+3, (c=-2 \wedge b=1))$

$\Leftrightarrow c=b=0 \vee (c=1 \wedge b=1)$

The strongest postconditions and weakest preconditions are related by several identities. See Identities 2.2.3, 2.2.4, and 2.2.5 in the proof section.

2.3. Stepwise Refinement

Just as a large program is impossible to comprehend as a whole, it is impractical to verify the correctness of a large program in one step. **Stepwise refinement** allows the verification of a large program to be broken up into many simpler verifications of separate sections of the program. Similarly, data refinement allows a program to be written and verified with high level data types and then be optimized to use machine-oriented data types. This combination is much simpler than verifying a program written directly in machine-oriented data types. The verification of the optimization is much easier than the original verification because we just have to show that the optimized program **simulates** the original, without understanding how the original program works. To use a familiar analogy, consider a compiler. A compiler translates a high level language into a more efficient machine language program without knowing how the high level program works.

Relatively recently [Back, 88], [Hehner et al., 86], and [Morris, 87b] have formalized the weakest precondition method into a programming calculus. They give a rigorous definition for **procedural refinement** of s to t, written $s \leq t$. (Procedural refinement is usually called refinement for brevity.)

Definition 2.3.1: Refinement

$$s \leq t \equiv [s.X \Rightarrow t.X]$$

Refinement is just replacement; this definition tells us when it is safe (correct) to replace statement s by statement t. Intuitively it ensures that t can be applied whenever s can be, and t accomplishes as much as s does. The easy way to remember it is “whatever s can do, t can do at least as well”. Notice the refinement relation is a second order logic expression because of the quantification over all predicates X. The brackets [ ] indicate the expression inside is universally quantified over all program variables.

The key to understanding stepwise refinement is to realize that a program can be thought of as a specification. For example, the program |[n := n+1; b := b+1]| specifies a program which increments n and b, and leaves all other variables unchanged. One program refines another when it satisfies the latter's specification. The only difference between a program and a specification is that a specification may contain statements that cannot be compiled. A program that contains the prescription statement is definitely a specification, because the prescription statement cannot be compiled, in general. Each prescription statement must be refined into a sequence of simple statements to convert a specification to a program. Looking at programs as specifications naturally leads to the idea of a wide spectrum language, which is **one** language used for specification, implementation, and all points between. See [Bauer et al., 89] and [Morris, 87a] for a full discussion.

There is a context sensitive way to express procedural refinement:

Theorem 2.3.2: Context Sensitive Refinement

$$\{P\}s\{?Q\} \leq t \equiv [P \wedge s.Q \Rightarrow t.Q]$$

P is true before s executes and Q is ALL that must be true after s executes. The difference between the two definitions is important: the first definition describes a context free refinement method, the second describes a context dependent method. Obviously whenever Theorem 2.3.1 is true, Theorem 2.3.2 will be true. If Theorem 2.3.1 is true then s can be replaced by t in any program in any context and the new program will behave the same as the old. With Theorem 2.3.2 s can safely be replaced by t only if P is true before s and the only purpose of s is to establish Q. Fewer refinements satisfy Theorem 2.3.1 than Theorem 2.3.2, but the refinements that satisfy

Theorem 2.3.1 are universally applicable while those that satisfy only Theorem 2.3.2 are usually only applicable to a specific program. In general we use Theorem 2.3.1, since Theorem 2.3.2 is just a special case of it.

Here is an example of refinement using Theorem 2.3.1:

$$s \leq (\text{if} \mid P1 \rightarrow \{P1\}s; \ \mid P2 \rightarrow \{P2\}s; \text{ fi})$$
$$\text{when } (P1 \vee P2)$$

We have created a program transformation rule. This is a useful refinement because we have reduced the problem of refining s to the two problems of refining s in context {P1} and in context {P2}. Once we show that this is a valid refinement, we can use it anytime, and our only proof obligation upon using it is to show $(P1 \vee P2)$ is true. We can prove that this refinement rule is valid:

Proof:

$s \leq (\text{if} \mid P1 \rightarrow \{P1\}s; \ \mid P2 \rightarrow \{P2\}s; \text{ fi})$

= { definition of refinement }

$\forall X:[s.X \Rightarrow (\text{if..fi}).X]$

= { definition of wp() of if }

$\forall X:[s.X \Rightarrow (P1 \vee P2) \wedge (P1 \Rightarrow s.X) \wedge (P2 \Rightarrow s.X)]$

= { assumption }

$\forall X:[s.X \Rightarrow (P1 \Rightarrow s.X) \wedge (P2 \Rightarrow s.X)]$

= { calculus }

True

Here is an example of a refinement that can be proven with Theorem 2.3.2.

$$b := 8 \ \{?\ b \text{ is even}\} \ \leq \ c := 13; \ b := (c+1) \ \{?\ b \text{ is even}\}$$

Proof:

$b := 8 \{?\ b \text{ is even}\} \;\le\; c := 13;\ b := (c+1)\ \{?\ b \text{ is even}\}$

= { def of context sensitive refinement }

$b{:=}8.\{b \text{ is even}\} \Rightarrow (c{:=}13;\ b{:=}c+1).\{b \text{ is even}\}$

= { definition of precondition for assignment}

$\{8 \text{ is even}\} \Rightarrow \{13 + 1 \text{ is even}\}$

=

TRUE

The idea behind stepwise refinement is to start with a program, e.g. |[s1;s2;s3]|, refine some subpart of it, e.g. s2 ≤ t2, to produce |[s1;t2;s3]| which is guaranteed to be a refinement of our original program. The fact that we can refine a subpart of a program and thus automatically refine the program is due to the monotonicity property of specifications:

$[X \Rightarrow Y] \Rightarrow [s.X \Rightarrow s.Y]$ for all specifications (programs) s.

Transitivity of refinement is expressed by this formula:

$(s1 \le s2) \wedge (s2 \le s3) \Rightarrow (s1 \le s3)$

Transitivity enables us to continue refining a program as long as we want, secure in the knowledge that the final program will be a valid refinement of our original program.

In summary, if statement s refines to t, then any program such as |[s1; s; s2; s3]| can be rewritten as |[s1; t; s2; s3]|, and the second program is as correct as the first.

2.4. Virtual Code

Virtual code is part of a program which is present during program development but missing in the final program. Obviously, code is only virtual if it can be deleted from the program without affecting observed behavior. It was first discussed by W. Polak [Polak, 81], who used it to facilitate verification of a compiler. Technically, a statement is virtual code if it refines to skip. Here is a program fragment that illustrates two types of virtual code:

```
|[
      integer i;
      if
            | (i = j)      → answer := 1;
            | (i = i+1) → j := j + 10;
      fi
      i := i + 3;
]|
```

The second branch of the if..fi is virtual code because the branch will never be taken. The i := i+3 statement is virtual code because the value of i is not referred to after the statement and before i disappears.

2.5. Saintly Nondeterminism

Consider the following program:

```
answer := 0;
while (answer = 0 ) do
      if
      | TRUE → answer := 1;
      | TRUE → answer := 2;
      | TRUE → skip;
      | TRUE → abort;
      fi
```

This program has many possible behaviors. Each time through the loop it nondeterministically chooses one of the four branches of the if..fi statement. If it takes the first or second choices it just exits the loop with answer set to 1 or 2. The third choice causes the loop to be repeated. The final choice causes the program to abort. The wp() of this program is FALSE, because there is no condition that can assure the successful termination of this program. If, however, this program behaves with **angelic nondeterminism** then the program would be guaranteed to terminate with answer = 1 or 2. An angelic program makes nondeterministic choices that lead to quick termination, if possible. In contrast, **demonic nondeterminism** makes nondeterministic choices which guarantee nontermination, if possible. If our program behaved with demonic nondeterminism it would abort because the fourth branch of the if..fi would be selected by the "demon".

We introduce the term **saintly nondeterminism**. Gries first used saintly nondeterminism, without calling it by name or generalizing the concept [Gries, 85]. A saintly program makes nondeterministic choices that lead to termination AND does its best to make sure that a program fulfills its obligations. For example, our sample program, if it behaves saintly, can assure that answer=1 when it terminates. Or if we prefer, the same program can assure that answer=2 when it terminates. In contrast, an angelic program could only assure that answer is 1 or 2 (we could not predict which). This is a strange concept because the behavior of a saintly statement depends on what it is used for.

For any statement s, we write s* for s operating saintly. The semantics for s* were essentially given by Gries:

Definition 2.5.1:

$$wp(s^*, X) = (s.T) \wedge \neg s.\neg X$$

$s.\neg X$ are all starting states for which s terminates in a state not satisfying X. $\neg s.\neg X$ are all states for which s does not terminate or s can possibly terminate in a state satisfying X. $(s.T) \wedge \neg s.\neg X$ are the states for which s terminates and can possibly terminate in a state satisfying X.

Later we will need the statement "b*:P", called the **saintly prescription** statement. Its weakest precondition, (b*:P).X, is defined as $(\exists b{:}P \wedge X)$. It is easy to show that b*:P is just the saintly equivalent of b:P, i.e. (b:P)*. Thus b*:P assigns to b the best value which makes P true. The saintly prescription may not be implementable in some cases. However, it is still useful during specification and refinement.

Dijkstra suggests four properties any weakest precondition definition should satisfy [Dijkstra, 76]. One is the law of monotonicity, which s* obeys:

Theorem 2.5.2:

$$[P \Rightarrow Q] \Rightarrow [s^*.P \Rightarrow s^*.Q]$$

Another is the conjunctive law, $[s.(P \wedge Q) = s.P \wedge s.Q]$, which s* only obeys in the left to right direction. Because s* only obeys the conjunctive law in one direction, it must be used with care.

Theorem 2.5.3:

$$[s^*.(P \wedge Q) \Rightarrow s^*.P \wedge s^*.Q]$$

s* obeys the disjunctive law $[s.(P \vee Q) \Rightarrow s.P \vee s.Q]$ in both directions:

Theorem 2.5.4:

$$[s^*.(P \vee Q) = s^*.P \vee s^*.Q]$$

The reader can check the intuitive fact $s \leq s^*$, i.e. making a statement saintly only increases its power. Also, if s is deterministic, $s = s^*$.

3. Data Refinement

The goal of this paper is to view data refinement as a special case of procedural refinement. With the formalisms described in the previous section we have the tools to proceed.

We state the general problem of data refinement as in [Morris, 89]. Consider three distinct sets of program variables b, k, and o. The abstract variables are b, the concrete variables are k, and the variables common to the abstract and concrete program are o. Data refinement is replacing a statement s which uses b and o by a statement t which uses k and o. (statement s and t may be arbitrarily large programs, not just primitive statements of the language.) Refining s to t effectively replaces the use of variables b with the use of new variables k.

In order to connect the sets of variables b, k, and o, we use a data invariant **I**. **I** can be any predicate relating b, k, and o. Of course, the choice of I is important, since a weak I may make it impossible to carry out the data refinement.

We distinguish two types of data refinements, nonlocal and local. Nonlocal data refinement, first identified here, is a generalization of local data refinement. A nonlocal data refinement replaces s in |[TypeB b; s1; s; s3;]| by a local block in k, |[TypeK k; t]|, to produce |[TypeB b; s1; |[TypeK k; t]|; s3;]|. The abstract variables b are created, destroyed, and possibly accessed outside of s. A local refinement replaces the local block |[TypeB b; s]| by the local block |[TypeK k; t]|. Abstract variables b are created upon entering s and destroyed upon leaving s, information we use to simplify the refinement process. Clearly, local data refinement is a special case of nonlocal refinement, since the local block |[TypeB b; s1]| is a special case of an arbitrary statement s.

We can characterize the difference between nonlocal and local data refinements in another way. A nonlocal data refinement represents a temporary change of a data structure, a data structure detour; a local data refinement represents a complete replacement of a data structure.

Consider a program which prints out paychecks based on employee data. The employee data might be represented in the program as a circular queue, allowing the program to search and insert easily. However, one part of the program might sort the employee data, a very inefficient operation on a circular queue. A programmer might decide to temporarily convert the employee data to an array of records during the sorting section of the program. He would use a nonlocal data refinement.

In contrast consider the example data refinement given at the beginning of the paper. We replaced the local variable s of type stack by s1 of type array, and s disappeared from the program. We made a local data refinement.

4. Theories of Data Refinement

Showing that a data refined program behaves the same as the original is the most important part of data refinement. In the literature are several logic formula that describe when correctness is preserved during data refinement. Here we describe each one and derive it as a special case of procedural refinement.

4.1. Morris' Data Refinement

Morris suggests the following in [Morris, 89], where X is any predicate using the variables of the abstract program, s:

Definition 4.1.1: Morris Data Refinement

$$\forall X:[(\exists b:I \wedge s.X) \Rightarrow t.(\exists b:I \wedge X)]$$

This mysterious looking equation is best understood in the context of procedural refinement, which is a much more intuitive concept. Remember, a program s refines to a program t, written $s \leq t$, when this formula is true:

$$\forall X:[s.X \Rightarrow t.X]$$

Since data refinement is just replacing one program by another, the refinement relation is exactly what we need: a condition that determines when it is correct to replace one program by another. Now we show how to view the complicated logic formula of Morris as a simple program refinement. Morris' equation actually is equivalent to the following refinement, where t is a program that does not use variables b, s is a program that does not use variables k, and I is the invariant that describes the relationship between b, k, and o:

Definition 4.1.2: ↓REFINEMENT

$b^*:I;\ s;\ \leq\ t;\ b^*:I$

↓REFINEMENT is pronounced "down-refinement". If this is translated to a logic formula, we get Morris' original equation.

↓REFINEMENT

=

$b^*:I;s\ \leq\ t;b^*:I$

= { definition of refinement }

$\forall X:[(b^*:I;\ s).X \Rightarrow (t;b^*:I).X]$

= { definition of b:*:I.X }

$\forall X:[(b^*:I).s.X \Rightarrow t.(\exists b:I \wedge X)]$

=

$\forall X:[(\exists b:I \wedge s.X) \Rightarrow t.(\exists b:I \wedge X)]$

=

Morris Data Refinement

We will use ↓REFINEMENT to refer to either the refinement equation or the logic equation: they are equivalent. An important feature of ↓REFINEMENT is **composability**. We can data refine a complex statement by data refining all its substatements. If s and t are actually composed of two programs in sequence, i.e. s = s1;s2 and t = t1;t2, then we want s to data refine to t when s1 data refines to t1 and s2 to t2. This works, as we can demonstrate:

Given:

$b^*:I;s1 \leq t1;b^*:I$

$b^*:I;s2 \leq t2;b^*:I$

$s = s1;s2$

$t = t1;t2$

We need to prove:

$b^*:I;s \leq t;b^*:I$

Proof:
We refine the left side to the right in steps.

    b*:I;s
= { def of s }
    b*:I;s1;s2
≤ { refining b*:I;s1 to t1;b*:I }
    t1;b*:I;s2
≤ { refining b*:I;s2 to t2;b*:I }
    t1;t2;b*:I
=
    t;b*:I

4.1.1. Using ↓REFINEMENT for Local Data Refinement

The previous section described how to derive Morris' data refinement relation from a simple procedural refinement. We now describe how ↓REFINEMENT is used to refine |[TypeB b; s]| to |[TypeK k; t]|, i.e. for local data refinement. We will actually refine |[TypeB b; s]| to |[TypeK k; k:(∃b:I); t]| with no loss of generality.

Assuming we know (b*:I;s ≤ t;b*:I), which is ↓REFINEMENT, the initial problem is that the s in |[TypeB b; s]| is not preceded by b*:I, so we cannot carry out the replacement indicated by ↓REFINEMENT. This is easily remedied, however, because we can refine the local block to one that also declares k (so we can use I in the local block), and then we can add b*:I before s. We write down the refinement and calculate what must be true for the refinement to be correct. See section 8 for the calculation of Lemma 4.1.1.1.

Lemma 4.1.1.1:
    |[TypeB b; s]| ≤ |[TypeB b; TypeK k; b*:I ; s]|
    if [∃b:I]

If we can prove [∃b:I], we can carry out this refinement. However, the condition is harder than it has to be. For example, if b and k are integers and I is (b/k = 2) than we could **not** carry out this refinement. If k is 0, there does not exist a b which makes I true. A better refinement first initializes k to make the refinement to b*:I easy. Then the refinement goes through without requiring any conditions to be checked.

Lemma 4.1.1.2:

$|[$ TypeB b; s $]| \le |[$ TypeB b; TypeK k; k:(∃b:I); b*:I; s $]|$
always.

Using Lemma 4.1.1.2 we can carry out the following refinement:

$|[$ TypeB b; s $]|$

≤ { Lemma 4.1.1.2 }

$|[$ TypeB b; TypeK k; k:(∃b:I) ; b*:I ; s $]|$

Since b*:I;s ≤ t;b*:I, and the left hand side appears in our program, we can now carry out the replacement:

$|[$ TypeB b; TypeK k; k:(∃b:I) ; b*:I ; s $]|$

≤ { ↓REFINEMENT }

$|[$ TypeB b; TypeK k; k:(∃b:I) ; t ; b*:I $]|$

Now, since t does not use b, and b cannot be accessed outside of the local block, the statements involving b are virtual code, and they can be deleted (technically they refine to skip):

$|[$ TypeB b; TypeK k; k:(∃b:I) ; t ; b*:I $]|$

≤

$|[$ TypeK k; k:(∃b:I) ; t $]|$

We have successfully data refined a local block using b to a local block using k, requiring only ↓REFINEMENT.

4.1.2. Using ↓REFINEMENT for Nonlocal Data Refinement

This section shows the problems with using ↓REFINEMENT for nonlocal data refinement. In a nonlocal data refinement we must replace s by a local block in k. We will actually refine s to |[TypeK k; k:I; t; b:Q]| in a number of steps, requiring ↓REFINEMENT and another condition. Notice that the complication of a nonlocal instead of a local refinement causes the local block in k to contain k:I instead of k:(∃b:I); also we have an additional b:Q statement. One may object to the b:Q statement on the grounds that it involves the abstract variable. This is necessary because to simulate s, which might change b, t would have to change b also. In general, it is impossible to refine s without having at least one assignment to b. One assignment, at the end of the block, seems the best one can do.

Just as in the local case, we must first prefix s by b*:I before we can apply ↓REFINEMENT.

Lemma 4.1.2.1:

s ≤ |[TypeK k; k:I; b*:I; s]| always.

Then we can use ↓REFINEMENT to refine further:

|[TypeK k; k:I; b*:I; s]|

≤ { ↓REFINEMENT }

|[TypeK k; k:I; t; b*:I]|

Now we are faced with an interesting dilemma. During local data refinement all the lines involving b become virtual code. During a nonlocal data refinement such as the one being developed here, the lines involving b are not virtual code; the code before s may set b, and the code after s may refer to b. But if b*:I cannot be deleted we have a problem: in general, b*:I cannot be implemented. Remember it is an example of saintly nondeterminism. It chooses among all values of b which make I true the best value, namely the value which makes the rest of the program terminate successfully. Although our data refinement guarantees that such a b value exists, in general there is no way to know **which** b value it is.

We can give two cases for which we can easily remove the saintly nondeterminism from b*:I. One case occurs when there is exactly one b which makes I true in the current context. We use the notation $(\exists!b:I)$ to mean there exists exactly one b such that I is true.

Theorem 4.1.2.2:

$\{P\}b^*:I \le \{P\}b:I \quad \text{if } [P \Rightarrow (\exists!b:I)]$

The other case that permits easy removal of the saintly nondeterminism occurs when we are doing a context sensitive refinement.

Theorem 4.1.2.3:

$\{P\}b^*:I\{?Q\} \le \{P\}b:Q\{?Q\} \quad \text{always}$

4.1.3. Piecewise Data Refinement

Morris also extended the idea of composability of a data refinement relation in [Morris, 89]. In a previous section we showed that a statement data refines if its substatements data refine. Morris extended this by giving simple formulas for data refining abstract statements, such as if..fi, do..od, or b := e, to analogous concrete statements. if..fi statements over b and o get converted to if..fi statements over k and o, do..od statements over b and o get converted to do..od statements over k and o, etc... As an example, we give his rule for converting an if..fi statement, assuming $s \ll t$ is shorthand for $(b^*:I;\ s \le t;\ b^*:I)$:

Theorem 4.1.3.1:
Let Q_i be $(\forall b{:}I \wedge G \Rightarrow P_i)$, $\{G \wedge P_i\}s_i \ll t_i$, and $[(\exists b{:}I \wedge G \wedge (\exists i{:}P_i)) \Rightarrow (\exists i{:}Q_i)]$.
Then

$$\{G\}\text{if} \mid P_i \rightarrow s_i \text{ fi} \ll \text{if} \mid Q_i \rightarrow t_i \text{ fi}$$

These rules are very handy in practice, because they reduce the hard problem of data refining a large program to many easier problems of data refining a single statement.

4.2. ↑REFINEMENT

One of the problems with ↓REFINEMENT appeared in our discussion of nonlocal data refinement. ↓REFINEMENT results in a statement involving saintly nondeterminism, which can only be easily implemented in special circumstances. Here we introduce another data refinement relation which does not share this disadvantage.

The new one is $b{:}I;s \leq t;b{:}I$. In a later section we will see this is the upward refinement relation suggested by He et al. [He et al., 86]. Though similar to ↓REFINEMENT in program form, the corresponding logic equation looks different:

DEFINITION 4.2.1: ↑REFINEMENT

$$b{:}I;s \leq t;b{:}I$$

=

$$\forall X{:}[(\forall b{:}I \Rightarrow s.X) \Rightarrow t.(\forall b{:}I \Rightarrow X)]$$

↑REFINEMENT is pronounced "up-refinement". ↑REFINEMENT has the same important composability feature as ↓REFINEMENT. Also, local data refinement works just as described for ↓REFINEMENT: |[TypeB b; s]| refines to |[TypeK k; k:(∃b:I); t]| when ↑REFINEMENT.

Nonlocal refinement using ↑REFINEMENT is easier to finish than with ↓REFINEMENT because we do not have to worry about refining a saintly statement at the end. Unfortunately, starting the refinement is harder.

$\{P\}s \le |[\ \text{TypeK } k;\ k{:}I;\ \{P\}b{:}I;\ s\ ]|$

= { proof omitted }

$\forall X{:}[P \wedge I \wedge s.X \Rightarrow (\forall b{:}I \Rightarrow s.X)]$

$\Leftarrow$

$[P \wedge I \Rightarrow (\exists! b{:}I)]$

Once we have refined from $\{P\}s$ to $|[\ \text{TypeK } k;\ k{:}I;\ \{P\}b{:}I;\ s\ ]|$, we can use ↑REFINEMENT to get to the following:

$|[\ \text{TypeK } k;\ k{:}I;\ t;\ b{:}I\ ]|$

We can leave the block this way because b:I does not involve saintly nondeterminism. To summarize, $\{P\}s$ refines to $|[\ \text{TypeK } k;\ k{:}I;\ t;\ b{:}I\ ]|$ if ↑REFINEMENT and $[P \wedge I \Rightarrow (\exists! b{:}I)]$.

4.3. Which is better?

One of the conclusions of this paper is that ↓REFINEMENT and ↑REFINEMENT account for all the data refinement theories in the literature. Are they actually different? In this section we show that some refinements are possible with ↓REFINEMENT but not possible with ↑REFINEMENT, and vice versa. We say that ↓REFINEMENT and ↑REFINEMENT are **incomparable**. Intuitively, the invariant, I, in ↑REFINEMENT is stronger than the I in ↓REFINEMENT. Although this means refinement takes place in a stronger context, it also means that the context is harder to establish. We then show the more dramatic fact that there are an infinite number of incomparable data refinement theories.

Example: This program fragment sets n to $z+2 \vee z+3$.

|[n = b+2; {?(n in z+2..z+3)}]|

We data refine integer b to another integer k, using the invariant I = (b = k-1..k). (b = k-1..k means that b is between k-1 and k inclusive.) Using ↑REFINEMENT we can data refine the fragment to a new program:

|[n = k+1; {?(n in z+2..z+3)}]|

This can be checked by plugging into ↑REFINEMENT:

↑REFINEMENT

=

$[(\forall b{:}I \Rightarrow (b \text{ in } z..z+1)) \Rightarrow (\forall b{:}I \Rightarrow (k \text{ in } z+1..z+2))]$

= { left hand side is k=z+1 }

TRUE

Using ↓REFINEMENT, in contrast, the data refinement is not allowed:

↓REFINEMENT

=

$[(\exists b{:}I \wedge (b \text{ in } z..z+1)) \Rightarrow (\exists b{:}I \wedge (k \text{ in } z+1..z+2))]$

= { left hand side is $(k=z \vee k=z+1 \vee k=z+2)$ }

FALSE

This happens because ↑REFINEMENT provides a stronger context for the data refinement than ↓REFINEMENT. ↑REFINEMENT assured us that k was z+1 before the statement; ↓REFINEMENT could only promise that k was $z \vee z+1 \vee z+2$.

We can give a contrasting example in which ↑REFINEMENT fails but ↓REFINEMENT succeeds. If I ≡ (b = k..(k+k/10)) then we can data refine |[b = b+50]| to |[k = k+50]| using ↓REFINEMENT but not using ↑REFINEMENT. The basic reason is that establishing the invariant after k=k+50 is harder with ↑REFINEMENT because ↑REFINEMENT is stronger than ↓REFINEMENT.

Of course, when faced with the opportunity to use two different refinements, the question is how to choose between them. For nonlocal refinements, $\uparrow$REFINEMENT is easier to finish but $\downarrow$REFINEMENT is easier to start. For local refinements, neither has an obvious advantage.

Actually, for most examples the choice of which to use seems to be irrelevant. If one works, the other usually works also. I is usually functional in k, specifically meaning $[(\exists b{:}I) \Rightarrow (\exists! b{:}I)]$, in which case both refinements are equivalent. Very roughly speaking, this is because $\uparrow$REFINEMENT makes sure that **every** b for a specific k is a "good" one; $\downarrow$REFINEMENT makes sure that **at least 1** b for a specific k is a "good" one; so if there is **exactly 1** b for a specific k then asking a question of **every** b will be the same as asking a question of **at least 1** b.

4.3.1. An Infinite Family of Data Refinement Theories

We can use the intuition of the previous paragraph to describe an infinite family of theories of data refinement. nRefinement defines a theory of data refinement parameterized by an integer n:

DEFINITION 4.3.1.1: nREFINEMENT

$$\forall X{:}[(\text{Exists at least n values of b such that } I \wedge s.X) \Rightarrow$$
$$t.(\text{Exists at least n values of b such that } I \wedge X)]$$

The case n=1 corresponds to $\downarrow$REFINEMENT. It is straightforward to show that each n generates a theory incomparable with a theory generated by a different n. Thus there are an infinite number of data refinement theories. Of course, most of these are not of practical interest.

4.4. Z Data Refinement

The Z specification language is becoming increasingly popular. [Hayes, 87] gives a good introduction to Z, [Spivey, 89] is a reference manual for Z. [Jones, 87] gives the data refinement formula for the Z language. [Neilson, 87] presents a simplified version. In this section we compare the Z rule with ↓REFINEMENT and ↑REFINEMENT. First we introduce the basics of Z, making some convenient changes of syntax from the "standard". Then we translate the Z results into weakest preconditions to make it easy to compare with the data refinement theories already discussed.

A program or statement, say OP1, is a predicate on the program variables c$bef (the state before executing the program) and c (the state after executing the program). For example, the square root function SQRT = ((d*d)=e$bef) ∧ (e=e$bef) sets d to the square root of e, and does not change e. The precondition *pre*OP1', expressing what OP1 expects to be true before it executes, is calculable directly from OP1:

Definition 4.4.1: preOP1'

$$preOP1' \equiv \exists c':OP1'$$

The result is a predicate on the program variables c because OP1' is shorthand for OP1[c'/c, c/c$bef]. We will use this shorthand extensively in what follows. In our SQRT example, *pre*SQRT = (∃e',d':SQRT') = e ≥ 0.

We will often need to rename the before and after variables of a predicate P. The notation e.P.f, or ePf, is a compact way to denote the renaming of the before and after variables to e and f respectively. Thus P' = c.P.c' = P[c'/c c/c$bef]. We also define P` = c$bef.P.c' = P[c'/c].

One of the simplest Z programs is S;T, which executes S and then executes T. Since S;T is a program it can be described by a Z predicate. There is a formula for calculating S;T from S and T, resulting in a new predicate in c$bef and c.

Definition 4.4.2: Z formula for S;T

S;T

=

$(\exists c':c\$bef.S.c' \wedge c'.T.c) \wedge (\forall c':c\$bef.S.c' \Rightarrow (\exists c:c'.T.c))$

Z does not normally include a notion of weakest precondition, but we specify one for convenience. Actually, we will use the weakest prespecification wps(), first described in [Hoare and He, 87c]. The expression wps(S, X) returns a predicate in c and c$bef which describes what must happen before executing S to establish X. It can be defined implicitly as the most general predicate Y for which Y;S terminates and Y;S $\Rightarrow$ X. Its explicit formula follows:

Definition 4.4.3: Weakest Prespecification

wps(OP1, X)

=

$(\exists c':OP1') \wedge (\forall c':OP1' \Rightarrow X`)$

If OP1 and X have different initial variable names, we can use the following simpler formula:

Definition 4.4.4: Weakest Prespecification

if T = c$Tbef.T.c$aft and X = c$Xbef.X.c$aft then

wps(T, X)

=

$(\exists c\$aft:T) \wedge (\forall c\$aft:T \Rightarrow X)$

The weakest prespecification is very similar to the weakest precondition. Here are some examples of its use. The program SKIP is just (c = c$bef). For any program S, wps(SKIP, S) = S, because if one wants SKIP to establish S, then S must be executed before SKIP: i.e. S;SKIP = S. Another example is wps(S, S), which simplifies to

SKIP: i.e. SKIP;S = S. For a final example, if X is (a = 2*a$bef + 2), and S is (a = 2*a$bef), then wps(S, X) is $(\exists a':a'=2*a) \wedge (\forall a':(a'=2*a) \Rightarrow a'=2*a\$bef+2)$ which simplifies to (a = a$bef + 1).

Using wps() we can derive the Z rule for procedural refinement of OP1 to OP2:

Theorem 4.4.5: Deriving Z Procedural Refinement

$$OP1 \leq OP2$$

=

$$\forall X:[wps(OP1, X) \Rightarrow wps(OP2, X)]$$

=

$$[preOP1' \Rightarrow preOP2'] \wedge$$
$$[preOP1' \wedge OP2' \Rightarrow OP1']$$

=

Z Procedural Refinement

Another important feature of wps() is the following theorem. Notice the similarity to the rule for wp(s;t, X).

Theorem 4.4.6: wps() of S;T

wps(S;T, X)

=

wps(S, wps(T, X))

We want to compare the Z data refinement formula to the data refinement theories already discussed. To this end we find a systematic way to translate the Z notation into our weakest precondition language. To convert a Z predicate OP1 to the language used previously we use a translation function g() with the following property:

Definition 4.4.7: Linking wps() and wp()

g(OP1) translates to program s

=

wps(OP1, X) = wp(s, X) for arbitrary X

The most important property of this definition is the following, which lets us translate the program S;T by translating each predicate independently.

Theorem 4.4.8: Translating S;T

S;T translates to g(S);g(T)

Now we can translate from Z procedural refinement to the weakest precondition refinement:

Theorem 4.4.9: Translating OP1 ≤ OP2

OP1 ≤ OP2

=

g(OP1) ≤ g(OP2)

Here is an explicit g() which satisfies the definition:

Theorem 4.4.10: An Explicit g()

g(OP1)

=

|[label$bef; {*pre*OP1'} c:OP1]|

4.4.1. The Z Data Refinement Formula

[Jones, 87] presents the Z data refinement formula without formal justification. [Josephs, 88] argues that procedural refinement is a special case of data refinement, which is exactly opposite the perspective presented in this paper. In any case, here is the Z formula for refining abstract operation S using abstract variables bbef, b, obef, o to concrete operation T using concrete variables kbef, k, obef, o with invariant I holding between b, k, and o:

Definition 4.4.1.1: Z Data Refinement

Specification S data refines to specification T under invariant I iff

(I ∧ *pre*S' ⇒ *pre*T') and

(I ∧ *pre*S' ∧ T' ⇒ (∃b':I' ∧ S'))

In the remainder of this section we derive the Z data refinement formula from the Z procedural refinement (II*;S) ≤ (T;II*), which lets us understand Z data refinement as

Z procedural refinement. Then we translate this refinement into our weakest precondition language to compare Z data refinement with the previous theories.

We can define the saintly predicate P* just as we defined the saintly statement. There is no explicit formula for P*, because its exact behavior depends on what it is used for. However, its wps() can be given:

Definition 4.4.1.2: Saintly Predicate

wps(P*, X)

=

(∃c':P' ∧ X`)

We define II as the analog of b:I. It is given by (I ∧ (o=o$bef) ∧ (k=k$bef)), a predicate which establishes I by changing only b. I is a predicate that does not contain any c$bef variables. II and II* translate to especially simple forms:

Lemma 4.4.1.3: Translating II

II translates to {∃b:I} b:I;

Lemma 4.4.1.4: Translating II*

II* translates to b*:I

Now we have all the tools to derive the Z formula for data refinement from the procedural refinement II*;S ≤ T;II*.

Theorem 4.4.1.5: Deriving Z Data Refinement

II*;S ≤ T;II*

=

Z Data Refinement Formula

Finally, we translate II*S ≤ T;II* in order to compare Z data refinement to the previous theories:

Theorem 4.4.1.6: Translating Z Data Refinement

$$II^*;S \leq T;II^*$$

=

$$b^*:I;s \leq t;b^*:I$$

where s is g(S) and t is g(T)

We see that Z data refinement is an instance of ↓REFINEMENT. In practice, Z data refinement and ↓REFINEMENT are essentially equivalent, because the restriction of s to g(S) is not a serious one.

4.4.2. Calculating the Refinement

As presented so far, the paradigm of data refinement is (1) choose an invariant I; (2) guess a concrete program t; (3) check that the refinement works by plugging s, t, and I into ↓REFINEMENT or ↑REFINEMENT. Josephs describes a way of calculating T directly from S and I. Actually, Hoare et al. first presented the idea of calculating the most general T that satisfies (II;S) ≤ (T;II) [Hoare et al., 87b]; Josephs' contribution was showing how to calculate the most general T that satisfies the Z data refinement formula [Josephs, 88], which we just showed is equivalent to finding the most general T that satisfies (II*;S) ≤ (T;II*). We restate his result:

Theorem 4.4.2.1: Calculating T

S data refines to X under I iff T procedurally refines to X and

$[I \wedge pre S' \Rightarrow pre T']$,

where T' is $(\exists b{:}I \wedge pre S') \wedge (\forall b{:}I \wedge pre S' \Rightarrow (\exists b'{:}I' \wedge S))$. In logic:

$$(II^*;S) \leq (X;II^*)$$

=

$$[I \wedge pre S' \Rightarrow pre T'] \;\; \wedge \;\; (T \leq X)$$

where T is $(\exists b{:}I \wedge pre S') \wedge (\forall b{:}I \wedge pre S' \Rightarrow (\exists b'{:}I' \wedge S))$

The $[I \wedge pre S' \Rightarrow pre T']$ term ensures such a T exists. Thus we have a formula for the most general concrete operation T data refining abstract operation S under invariant I.

The new paradigm of data refinement is (1) choose an invariant I; (2) calculate the most general T that refines S under invariant I; (3) implement T.

In the general case calculating the refinement T is not as impressive as it sounds. The formula for T given above is really the same as ↓REFINEMENT converted into an explicit predicate. When using weakest preconditions, a procedure can be specified by stating what is true before it executes and what must be true after it executes. ↓REFINEMENT = $[(\exists b{:}I \wedge s.X) \Rightarrow t.(\exists b{:}I \wedge X)]$ says exactly this: $(\exists b{:}I \wedge s.X)$ is true before t executes, $(\exists b{:}I \wedge X)$ must be true after. More impressive is Morris' idea of extending composability of data refinement to all statements of the language.

4.5. Hoare's ↑REFINEMENT

[Hoare et al., 87b] takes the same approach to data refinement as this paper: data refinement is just a special case of procedural refinement.

For simplicity of description and use, Hoare et al. make the assumption that all programs are total, i.e. defined on all inputs. We generalize his technique to partial programs to facilitate comparison with the other theories. In addition he uses the relational formalism of computing. To avoid introducing another formalism in our paper, we will translate his results into predicate (Z) notation as in the previous section.

Remember that we derived ↑REFINEMENT from the weakest precondition equation $b{:}I;s \leq t;b{:}I$. [Hoare et al., 87b] suggests calculating from a similar equation, $II;S \leq T;II$, the most general T which satisfies it for a given S and II. This leads naturally to the invention of the weakest prespecification, wps(), which was introduced in the previous section. They use the notation Q\P as a compact way to write wps(Q, P). The following theorem says that of all the T for which P refines to T;Q, T = Q\P is the most general.

Definition 4.5.1: Weakest Prespecification Q\P

$$(Q\backslash P \le T) \wedge [\mathit{pre}P' \Rightarrow \mathit{pre}(Q\backslash P)'] \Leftrightarrow (P \le T;Q)$$

The explicit formula for Q\P was given in the previous section. Plugging the explicit formula into Definition 4.5.1 shows its correctness. The $[\mathit{pre}P' \Rightarrow \mathit{pre}(Q\backslash P)']$ assures that a T exists. If it is false, no such T exists.

Now that the weakest prespecification can be calculated, data refinement is trivial. We want the most general T such that II;S ≤ T;II, namely:

$$T = (II\backslash II;S) = \mathrm{wps}(c.II.c', c\$bef.(II;S).c')$$
$$= (\exists c':c.II.c') \wedge (\forall c':c.II.c' \Rightarrow c\$bef.(II;S).c')$$

T is the most general specification of a program which refines II;S to T;II, if such a program exists. By looking at the formula for T one can understand how T can fail to exist. T basically excludes starting states if they can lead to a result differing from a possible result of S. If T excludes too many starting states then T will not be able to start executing in all the states that II;S can. This is what $[\mathit{pre}(II;S)' \Rightarrow \mathit{pre}T']$ tests. Remembering the initial discussion of refinement, the test checks that T is **as defined** as II;S.

We want to translate Hoare's ↑REFINEMENT into the weakest precondition language so that it can be compared to the previous theories. The next theorem carries out the translation.

Theorem 4.5.2: Translating II;S $\leq$ T;II

II;S $\leq$ T;II

=

$\{\exists b{:}I\}b{:}I;s \leq t;\{\exists b{:}I\}b{:}I$

where s is g(S) and t is g(T)

We see that Hoare's ↑REFINEMENT is **almost** the same as ↑REFINEMENT. The difference is that b:I is only used when (∃b:I) is known. It is easy to show that Hoare's ↑REFINEMENT is incomparable with ↑REFINEMENT. In practice, though, they are essentially equivalent. It is rare to use b:I in a context where (∃b:I) is false. ↑REFINEMENT is usually preferable because its formula is simpler.

4.6. Hoare's ↓REFINEMENT

[Hoare et al., 87b] points out another data refinement relation close in appearance to II;S $\leq$ T;II. The new one is S;JJ $\leq$ JJ;T, where this time JJ is a relation from abstract variables to concrete variables that assigns only to k to make I true. This formula translates to $s;\{\exists k{:}I\}k{:}I \leq \{\exists k{:}I\}k{:}I;t$. By the same argument used in the last section, we can use instead the simpler formula $s;k{:}I \leq k{:}I;t$. One thing to notice is that the right side does not assign to b. As such, in general the refinement will not be applicable very often. Intuitively, the right side has to simulate the left side, so if s assigns to b, t would have to also. This problem can be fixed by appending {b?TRUE} to both sides. In a later section we will be able to prove the following surprising result:

Theorem 4.6.1:

$s;\ k{:}I;\ \{b?TRUE\} \leq k{:}I;\ t;\ \{b?TRUE\}$

=

$b^*{:}I;\ s;\ \{k?TRUE\} \leq t;\ b^*{:}I;\ \{k?TRUE\}$

=

↓REFINEMENT

Thus we now have two different characterizations of ↓REFINEMENT, one involving saintly nondeterminism, the other not. Further, we conclude that Hoare's ↓REFINEMENT is essentially the same as ↓REFINEMENT.

4.7. Gries and Prins

Gries and Prins introduced a strange looking theory of data refinement [Gries and Prins, 85]. They state the problem slightly differently than we have previously. Besides statements s and t and variables k, b, and o, they introduce t', which is the same as t but with variables o renamed to variables o'. Thus t' and s have no variables in common.

Gries and Prins suggest the relation $[I \wedge s.T \wedge o{=}o' \Rightarrow t'.\neg s.\neg(I \wedge o{=}o')]$. We abbreviate this definition as **?REFINEMENT**. Our main result follows:

Theorem 4.7.1:

$b^*{:}I;s \leq t;b^*{:}I$ if

$[I \wedge s.T \wedge o{=}o' \Rightarrow t'.\neg s.\neg(I \wedge o{=}o')]$ (?REFINEMENT)

Thus ↓REFINEMENT is at least as general as ?REFINEMENT. Now we show how to derive ?REFINEMENT using procedural refinement, and from this we show the truth of Theorem 4.7.1.

We can understand ?REFINEMENT by noticing that $\neg s.\neg(I \wedge o{=}o')$, also written $\neg wp(s, \neg(I \wedge o{=}o'))$, is exactly $wp(s^*, I \wedge o{=}o')$. This suggests the procedural refinement $\{I \wedge o{=}o'\}s \leq t';s^*;\{I \wedge o{=}o'\}$. We can give an explanation of the intuition behind $t';s^*;\{I \wedge o{=}o'\}$. We imagine t' and s running in parallel, with t' simulating s. Since they have no variables in common, though, we only require that $(I \wedge o{=}o')$ be true before and after. More specifically, if t' and s are run in parallel, when they terminate we want b and k to be related by I and the o' variable equal to a possible value of o. Since t' and s share no variables in common, running them sequentially is like running them in parallel; making s behave saintly lets it make nondeterministic decisions which cause o to match o' and I to be true.

We convert $\{I \wedge o=o'\}s \le (t';s^*;\{I \wedge o=o'\})$ into logic to show that it derives ?REFINEMENT.

$$
\begin{aligned}
&\quad \{I \wedge o=o'\}s \le (t';s^*;\{I \wedge o=o'\}) \\
&= \{ \text{ def of refinement } \} \\
&\quad (\forall X:[I \wedge o=o' \wedge s.X \Rightarrow (t';s^*).(I \wedge o=o')] \;\wedge \\
&\quad [I \wedge o=o' \wedge s.X \Rightarrow (t';s^*).X]) \\
&= \{ \text{ def of wp() } \} \\
&\quad (\forall X:[I \wedge o=o' \wedge s.X \Rightarrow t'.\neg s.\neg(I \wedge o=o')] \;\wedge \\
&\quad [I \wedge o=o' \wedge s.X \Rightarrow s.T] \;\wedge \\
&\quad [I \wedge o=' \wedge s.X \Rightarrow t'.\neg s.\neg X]) \\
&= \{ \neg s.\neg X \text{ does not contain concrete vars, so } t'.\neg s.\neg X = (\neg s.\neg X) \wedge t'.T \} \\
&\quad (\forall X:[I \wedge o=o' \wedge s.X \Rightarrow t'.\neg s.\neg(I \wedge o=o')] \;\wedge \\
&\qquad [I \wedge o=o' \wedge s.X \Rightarrow s.T] \;\wedge \\
&\qquad [I \wedge o=o' \wedge s.X \Rightarrow \neg s.\neg X] \;\wedge \\
&\qquad [I \wedge o=o' \wedge s.X \Rightarrow t'.T]) \\
&= \{ s.X \Rightarrow \neg s.\neg X \text{ and } s.T,\ t'.\neg s.\neg(I \wedge o=o') \Rightarrow t'.T \} \\
&\quad (\forall X:[I \wedge o=o' \wedge s.X \Rightarrow t'.\neg s.\neg(I \wedge o=o')]) \\
&= \{ \text{ TRUE for X } \} \\
&\quad [I \wedge o=o' \wedge s.T \Rightarrow t'.\neg s.\neg(I \wedge o=o')])
\end{aligned}
$$

We can show that ↓REFINEMENT is at least as general as ?REFINEMENT. For any reasonable language, any program s or s* can be refined by all*:T, which is a saintly nondeterministic assignment to all the program variables. (We gloss over details of input/output, which can be handled with some simple extensions.). Then it is easy to show that $|[\ s^*;\ \{I \wedge o=o'\}\ ]| \le |[\ o:=o';\ b^*:I\ ]|$.

Also, $|[\ t';\ o:=o'\ ]|.X \Rightarrow t.X$ if X does not contain o'. Thus for our purposes $|[\ t';\ o:=o';\ ]| \le t$. Now we can show ?REFINEMENT implies ↓REFINEMENT:

Proof of Theorem 4.7.1:

?REFINEMENT

=

$\{I \wedge o=o'\}s \leq (t';s^*;\{I \wedge o=o'\})$

$\Rightarrow$ { |[s^*; $\{I \wedge o=o'\}$]| $\leq$ |[o:=o'; b*:I]| }

$\{I \wedge o=o'\}s \leq (t'; o:=o'; b^*:I)$

$\Rightarrow$ { |[t'; o:=o';]| $\leq$ t }

$\{I\}s \leq (t; b^*:I)$

$\Rightarrow$ { prefix both sides with b*:I, simplify }

↓REFINEMENT

We have shown that ↓REFINEMENT is more general than ?REFINEMENT: notice that the first implication in the proof of Theorem 8 only goes in one direction. However, ?REFINEMENT has the desirable feature of getting rid of the universal quantification over predicates X. (In the next section we do the same for ↓REFINEMENT.) On the other hand ?REFINEMENT requires a cumbersome renaming procedure and a potentially complicated weakest precondition calculation.

5. Generalizing and Specializing

In the previous section several theories of data refinements were presented. Each was shown to be essentially an instance of $\downarrow$REFINEMENT or $\uparrow$REFINEMENT. Here we extend data refinement to the case where the I at the start is different from I at the end; we summarize the formulas for nonlocal and local refinement using $\downarrow$REFINEMENT and $\uparrow$REFINEMENT; and we present a few important special cases for each. The weakest precondition formalism will be used, and we present formulas which do not involve quantification over all predicates.

Morris characterized refinement using this formula:

$$s \leq t \;=\; \forall X{:}[s.X \Rightarrow t.X]$$

In practice, using this formula is difficult, because there is no easy way to prove a formula for all possible predicates X. Fortunately, [Back, 88] showed that this definition is equivalent to the following formula, which does not involve quantification over predicates:

Theorem 5.1:

$$s \leq t \;=\; [c\emptyset \wedge s.T \Rightarrow t.sp(s,\, c\emptyset)]$$

The cø term is shorthand for a predicate ($b\emptyset=b \;\wedge\; o\emptyset=o \;\wedge\; k\emptyset=k$). bø, oø, and kø are new variables corresponding to b, k, and o. They are used to remember the initial values of the program variables. There is an assumption that s and t do not alter the variables kø, bø, or oø. We also use the predicates kø for ($k\emptyset=k$), bø for ($b\emptyset=b$), and oø for ($o\emptyset=o$). In this paper we always use $c\emptyset = (b\emptyset=b \;\wedge\; o\emptyset=o)$, since these are all the abstract program variables.

Using Theorem 5.1 it is easy to prove Theorem 4.6.1. Although Theorem 5.1 tells how to test for refinement between any two programs, in practice it is hard to use. The main problem is calculating t.sp(s, cø). For even programs as small as a page long, the logic formula for t.sp(s, cø) can be unmanageable. (One proof obligation calculated by the IOTA system [Nakajima and Yuasa, 83] was 89 lines, but the program and specification were only 20 lines!) ↓REFINEMENT and ↑REFINEMENT are composable, fortunately. Instead of calculating sp(s, cø) for a large program s, the problem can be broken into many calculations of sp(s_i, cø) for substatements s_i. Unfortunately, during nonlocal data refinement an sp(s, cø) term must be calculated before ↓REFINEMENT or ↑REFINEMENT can be applied. We give a number of special cases for the purpose of avoiding this computation.

5.1. Data Refinement Using Semi-Invariants

In this section we generalize ↓REFINEMENT and ↑REFINEMENT to use invariant I1 at the start of the data refinement and I2 at the end. Also we give an explicit formula for the two refinement relations. Data refinement with a different I at the start and end is called "data refinement using **semi-invariants**". (Since I changes from start to finish, calling it an invariant is no longer accurate.) In practice changing the invariant is very useful. Often, for example, one wants to temporarily alter an invariant. An example of this appears at the end of the paper.

In the following, the cø term is shorthand for a predicate (bø=b $\wedge$ oø=o). bø, oø, and kø are new variables corresponding to b, k, and o. They are used to remember the initial values of the program variables. There is an assumption that s and t do not alter the variables kø, bø, or oø. We also use the predicates kø for (kø=k), bø for (bø=b), and oø for (oø=o).

Theorem 5.1.1:

$$s;\ k{:}I2;\ \{b?TRUE\} \le k{:}I1;\ t;\ \{b?TRUE\}$$

=

$$b^*{:}I1;\ s;\ \{k?TRUE\} \le t;\ b^*{:}I2;\ \{k?TRUE\}$$

=

$$[c\emptyset \wedge I1 \wedge s.T \Rightarrow t.(\exists b{:}I2 \wedge sp(s,c\emptyset))]$$

=

$\downarrow$REFINEMENT

Theorem 5.1.2:

$$b{:}I1;\ s;\ \{k?TRUE\} \le t;\ b{:}I2;\ \{k?TRUE\}$$

=

$$[k\emptyset \wedge o\emptyset \wedge (\forall b{:}I1 \Rightarrow s.T) \Rightarrow t.(\forall b{:}I2 \Rightarrow sp(s,\ I1[k\emptyset/k] \wedge o\emptyset))]$$

=

$\uparrow$REFINEMENT

From now on $\downarrow$REFINEMENT and $\uparrow$REFINEMENT refer to these generalized definitions. We introduce the notation s $\ll_{\mathbf{DN}}$ t as shorthand for $\downarrow$REFINEMENT holding between s, t, I1, and I2. Similarly, s $\ll_{\mathbf{UP}}$ t is shorthand for $\uparrow$REFINEMENT holding between s, t, I1, and I2.

An important simplification occurs when s is of the form s';{?Q}. Then sp(s,..) does not have to be calculated at all, because we can use the context sensitive refinement rule, Theorem 2.3.2.

Theorem 5.1.3:

$$s';\{?Q\} \ll_{\mathbf{DN}} t$$

=

$$[I1 \wedge s'.Q \Rightarrow t.(\exists b{:}I2 \wedge Q)]$$

Theorem 5.1.4:

$$s';\{?Q\} \ll_{\mathbf{UP}} t$$

=

$$[(\forall b{:}I1 \Rightarrow s'.Q) \Rightarrow t.(\forall b{:}I2 \Rightarrow Q)]$$

The most important property of the two data refinements is composability. Here are two composability theorems:

Theorem 5.1.5:
if s1 $\ll_{\mathbf{DN}}$ t1 with invariants I1 and I2, and
s2 $\ll_{\mathbf{DN}}$ t2 with invariants I1' and I2', then
s1;s2 $\ll_{\mathbf{DN}}$ t1;t2 with invariants I1 and I2' if
$[I2 \Rightarrow I1']$.

Theorem 5.1.6:
if s1 $\ll_{\mathbf{UP}}$ t1 with invariants I1 and I2, and
s2 $\ll_{\mathbf{UP}}$ t2 with invariants I1' and I2', then
s1;s2 $\ll_{\mathbf{UP}}$ t1;t2 with invariants I1 and I2' if
$[I2 \Rightarrow I1']$.

5.2. Nonlocal Refinement: Summary and Special Cases

A nonlocal data refinement refines from s to |[TypeK k; k:I1; t; b:P]|. We have the following general result:

Theorem 5.2.1: Nonlocal Data Refinement using ↓REFINEMENT:
$s \le$ |[TypeK k; k:I1; t; $\{\exists b{:}I2 \wedge sp(s, c\emptyset)\}$ b:P]|
if $[c\emptyset \wedge s.T \Rightarrow s.(\forall k{:}I2 \Rightarrow (\forall b{:}P \Rightarrow sp(s, c\emptyset)))]$ and
s $\ll_{\mathbf{DN}}$ t

The first proof obligation has the troublesome sp(s, cø) term, which in practice is difficult to compute. We find several cases for which we can avoid the computation. The first special case does not avoid the computation, but it prepares the way for other special cases. The second takes advantage of context sensitive refinement.

Corollary 5.2.2: Special Case of sp(s, cø) for P:
$s \le$ |[TypeK k; k:I1; t; $\{\exists b{:}I2 \wedge sp(s, c\emptyset)\}$ b:sp(s, cø)]|
if s $\ll_{\mathbf{DN}}$ t

Corollary 5.2.3: Special Case of s of the form s';{?Q} :
s';{?Q} $\le$ |[TypeK k; k:I1; t; $\{\exists b{:}I2 \wedge Q\}$ b:Q]|
if s';{?Q} $\ll_{\mathbf{DN}}$ t

Often, the relation I has the feature that for any k there is at most one b which makes I true. This special case makes it easy to eliminate the sp(s, cø) term:

Corollary 5.2.4: Special Case of $[(\exists b{:}I2 \wedge Q) \Rightarrow (\exists! b{:}I2 \wedge Q)]$:

$s\{Q\} \leq |[$ TypeK k; k:I1; t; $\{\exists b{:}I2 \wedge Q\}$ b:I2 $\wedge$ Q $]|$
if s{Q} **‹‹DN** t

The next set of theorems are for doing nonlocal refinement using ↑REFINEMENT.

Theorem 5.2.5: Nonlocal Data Refinement Using ↑REFINEMENT:

$s \leq |[$ TypeK k; k:P; t; b:I2 $]|$
if $[c\emptyset \wedge s.T \wedge P \Rightarrow (\forall b{:}I1 \Rightarrow s.sp(s, c\emptyset))]$ and
s **‹‹UP** t

Corollary 5.2.6: Special Case P is I1', I1 is $b=b\emptyset \wedge I1'$

$s \leq |[$ TypeK k; k:I1'; t; b:I2 $]|$
if s **‹‹UP** t with $I1 = b=b\emptyset \wedge I1'$

Corollary 5.2.7: Special Case s is of the form {b?Q}s',
P is $(\exists b{:}I1' \wedge Q)$, I1 is $I1' \wedge Q$

$\{b?Q\}s' \leq |[$ TypeK k; k:$(\exists b{:}Q \wedge I1')$; t; b:I2 $]|$
if {b?Q}s **‹‹UP** t with $I1 = I1' \wedge Q$

Using special case 5.2.7, the following theorem is useful:

Theorem 5.2.8:

{b,c?P} = {b?P}{b,c?P}

Corollary 5.2.9: Special Case if $[(\exists b{:}I1' \wedge Q) \Rightarrow (\exists! b{:}I1' \wedge Q)]$

$\{Q\}s \leq |[$ TypeK k; k:I1'; t; b:I2 $]|$
if {Q}s **‹‹UP** t with $I1 = I1' \wedge Q$

5.3. Local Refinement: Summary

Theorem 5.3.1: Local Data Refinement with ↓REFINEMENT

|[TypeB b; s]| ≤ |[TypeK k; k:(∃b:I1); t; {∃b:I2}]|
if s ‹‹**DN** t

Theorem 5.3.2: Local Data Refinement with ↑REFINEMENT

|[TypeB b; s]| ≤ |[TypeK k; k:(∃b:I1); t; {∃b:I2}]|
if s ‹‹**UP** t

6. Using Data Refinement for Optimization

Earlier we described the importance of data refinement for changing from inefficient abstract data types to efficient concrete data types. Here we point out that many procedural optimizations can be treated as data refinements.

Ordinarily, proving that one chunk of a program optimizes to another involves proving that the first chunk refines to the second. One could use Theorem 5.1, but that would involve calculating the wp() and sp() of a potentially large program. Another approach would be to use a library of correctness preserving program transformations. But research into this approach has not identified a suitably finite set of transformations. Instead, using data refinement and the composability of ↓REFINEMENT and ↑REFINEMENT, one can break the refinement into a number of small refinements, linked by potentially different invariants.

As an example we carry out an optimization on a sorting program. This problem also serves as an application of the theories presented in this paper. The reader will doubtless notice that even this seemingly simple problem is difficult to do by hand: machine support is crucial.

Example: this program is a simple insert sort developed by stepwise refinement. The invariants used in deriving the program have been left out, as they are not used in the data refinement demonstrated.

```
TypeInt n;
Type array[1..n] of int a[];
{n >= 2}

|[ TypeInt p;

p := 1;
while p ≠ n do
      begin
      p := p+1;
      |[TypeInt q;

      q := p;
*     while (q ≠ 1) & (a[q] < a[q-1]) do
*           begin
**          |[ TypeInt temp;
**          temp := a[q];
**          a[q] := a[q-1];
**          a[q-1] := temp;
**          ]|
**          q := q-1;
*           end
      ]|
      end
]|
```

Looking at the inner loop, marked with stars, we notice we can optimize it by holding a[q] in a temporary variable while we shift it into place. This procedural refinement could be done in a number of relatively complicated program transformations. Instead we treat it as a nonlocal data refinement using the invariant I:

$$I \equiv (\forall i: (i \neq q) \Rightarrow a'[i] = a[i]) \wedge (temp' = a[q])$$

Here the abstract variables are a[], and the concrete variables are a'[] and temp'. We will use a nonlocal refinement with ↑REFINEMENT. For any a'[] and temp' there is only one a[] which makes I true, so we can use Corollary 5.2.9. Calling the inner loop s, Corollary 5.2.9 tells us:

s ≤ |[TypeArray a'; TypeInt temp'; a',temp':I; t; a:I]|

if s ‹‹**UP** t with I1 = I2 = I.

Refining “a',temp':I” is easy:

a',temp':I ≤ a'[] := a[]; temp' := a[q];

Refining "a:I" is also easy:

a:I $\leq$ a[] := a'[]; a[q] := temp';

Using the composability of ↑REFINEMENT we can data refine each statement in turn. We only show the refinement explicitly for the 6 lines marked with 2 stars which decrement q and swap a[q] with a[q-1]. This refinement will involve two different invariants: I1 = I and I2 = I[q-1/q]. We need to do this because our invariant is not reestablished until q is decremented. When we refine the line q := q-1 we will use I1 = I[q-1/q] and I2 = I, reestablishing the original invariant.

s.T is TRUE;

I[kø/k] is $(\forall i: (i \neq q) \Rightarrow aø'[i] = a[i]) \wedge (tempø' = a[q])$

I1 $\equiv$ I is $(\forall i: (i \neq q) \Rightarrow a'[i] = a[i]) \wedge (temp' = a[q])$

I2 $\equiv$ I[q-1/q] is $(\forall i: (i \neq q-1) \Rightarrow a'[i]=a[i]) \wedge (temp' = a[q-1])$

sp(|[...]|, I[kø/k] $\wedge$ oø)
= { sp for |[..]| }
(∃temp:sp(a[q-1] := temp;, sp(a[q] := a[q-1];, sp(temp := a[q],
$(\forall i: (i \neq q) \Rightarrow aø'[i] = a[i]) \wedge (tempø' = a[q]) \wedge oø))))$
= { sp for assignment }
(∃temp:sp(a[q-1] := temp;, sp(a[q] := a[q-1];, (temp=a[q]) $\wedge$
$(\forall i: (i \neq q) \Rightarrow aø'[i] = a[i]) \wedge (tempø' = a[q]) \wedge oø)))$
= { sp for assignment }
(∃temp, i1:sp(a[q-1] := temp;, (a[q]=a[q-1]) $\wedge$ (temp=i1) $\wedge$
$(\forall i: (i \neq q) \Rightarrow aø'[i] = a[i]) \wedge (tempø' = i1) \wedge oø))$
= { temp for i1, sp for assignment }
(∃temp, i2:(a[q-1]=temp) $\wedge$ (a[q]=i2) $\wedge$
(aø'[q-1]=i2) $\wedge$ $(\forall i: (i \neq q,q-1) \Rightarrow aø'[i]=a[i]) \wedge (tempø'=temp) \wedge oø)$
= { tempø' for temp, aø'[q-1] for i2 }
(a[q-1]=tempø') $\wedge$ (a[q]=aø'[q-1]) $\wedge$ $(\forall i: (i \neq q,q-1) \Rightarrow aø'[i]=a[i]) \wedge oø$

Plugging in to ↑REFINEMENT we get the following:

$$[(a'=a\varnothing') \wedge (temp'=temp\varnothing') \wedge (p=p\varnothing) \wedge (n=n\varnothing) \wedge o\varnothing \Rightarrow$$
$$t.(\forall a{:}I2 \Rightarrow (a[q-1]=temp\varnothing') \wedge (a[q]=a\varnothing'[q-1]) \wedge o\varnothing \wedge$$
$$(\forall i: (i\neq q,q-1) \Rightarrow a\varnothing'[i]=a[i]))$$

= { plugging in I2, simplifying }

$$[(a'=a\varnothing') \wedge (temp'=temp\varnothing') \wedge (p=p\varnothing) \wedge (n=n\varnothing) \wedge o\varnothing \Rightarrow$$
$$t.((temp'=temp\varnothing') \wedge (a'[q]=a\varnothing'[q-1]) \wedge o\varnothing \wedge$$
$$(\forall i: (i\neq q,q-1) \Rightarrow a\varnothing'[i]=a'[i]))$$

The postcondition of t says that everything is unchanged, except for a'[q] and a'[q-1]. The new value of a'[q-1] can be anything and the new value of a'[q] must be the old value of a'[q-1]. Thus the t we need is obvious:

a'[q] := a'[q-1];

Plugging it in proves that it works. Continuing in this way, and collecting the various refinements, the data refined inner loop looks as follows:

```
|[ TypeArray a'; TypeInt temp';

      a'[] := a[];
      temp' := a[q];

      while (q ≠ 1) & (temp' < a'[q-1]) do
            begin
            a'[q] := a'[q-1];
            q := q-1;
            end

      a[] := a'[];
      a[q] := temp';
]|
```

Renaming a'[] to a[] and temp' to temp (which is technically a nonlocal data refinement with abstract variables a' and temp', concrete variable temp, common variables a and q, and invariant I being (a'=a ∧ temp'=temp)), we end up with our final program. Notice the assignments a'[]:=a[] and a[]:=a'[] become virtual code and disappear:

```
TypeInt n;
Type array[1..n] of int a[];
{n >= 2}

|[ TypeInt p;

p := 1;
while p ≠ n do
      begin
      p := p+1;
      |[ TypeInt q,temp;

      q := p;
      temp := a[q];

      while (q ≠ 1) & (temp < a[q-1]) do
            begin
            a[q] := a[q-1];
            q := q-1;
            end

      a[q] := temp;
      ]|
      end
]|
```

7. Summary

Data refinement is crucial for stepwise refinement of programs. When data refinement is thought of as a special case of procedural refinement, it becomes more understandable. Using this approach, two different theories of data refinement account for all the theories in the literature. We demonstrated that there are actually an infinite number of data refinement theories.

Besides making clear the similarities of the various data refinement theories, this paper introduces nonlocal data refinement, refinement with semi-invariants, and the idea of using data refinement for procedural optimization. In addition, a number of important special cases are identified, and closed formulas given, that make data refinement more practical. Finally, two formalisms are described which prove useful in data refinement and should have application to procedural refinement: reassertion statements and saintly nondeterminism.

8. Proofs

The following laws are used throughout (from [Morris, 89]):

predicate calculus laws:

$$(P \wedge (Q \Rightarrow R)) = (P \wedge (P \wedge Q \Rightarrow R)) = (P \wedge (Q \Rightarrow P \wedge R))$$

$$((P \vee Q) \Rightarrow R) = ((P \Rightarrow R) \;\wedge\; (Q \Rightarrow R))$$

$$(P \Rightarrow Q \wedge R) = ((P \Rightarrow Q) \;\wedge\; (P \Rightarrow R))$$

importation/exportation:

$$[(P \wedge Q \Rightarrow R) = (P \Rightarrow (Q \Rightarrow R))]$$

absorption:

$$[(\forall x{:}P)] = [P]$$

$[(\forall x{:}P \Rightarrow Q)] = [((\exists x{:}P) \Rightarrow Q)]$ for Q not free in x.

$[(\forall x{:}P \Rightarrow Q)] = [(P \Rightarrow (\forall x{:}Q))]$ for P not free in x.

1-point:

$$[(\exists x{:}x{=}y \;\wedge\; P) = P[y/x]|$$

$$[(\forall x{:}x{=}y \Rightarrow P) = P[y/x]|$$

splitting:

$$[P \Rightarrow Q \wedge R] = [P \Rightarrow Q] \;\wedge\; [P \Rightarrow R]$$

adding:

$$[(\exists x{:}P) \wedge (\forall x{:}Q) = (\exists x{:}P \wedge Q) \wedge (\forall x{:}Q)]$$

Identity 2.2.3: Identity between wp(), sp()
$[P \wedge s.T \Rightarrow s.sp(s, P)]$
(from [Back, 88])

Identity 2.2.4: Identity between wp(), sp()
$[sp(s, s.P) \Rightarrow P]$
(from [Back, 88])

Identity 2.2.5: Identity between wp(), sp()

$[c\emptyset \wedge s.X \Rightarrow (\forall c{:}sp(s, c\emptyset) \Rightarrow X)]$

where c represents all the program variables b,k,o,

$c\emptyset$ is the predicate $(b=b\emptyset) \wedge (k=k\emptyset) \wedge (o=o\emptyset)$

for new variables $b\emptyset, k\emptyset, o\emptyset$.

TRUE

=

$[c\emptyset \wedge s.X \Rightarrow (\forall c{:}c\emptyset \Rightarrow s.X)]$

$\Rightarrow$ { monotonicity of s }

$[c\emptyset \wedge s.X \Rightarrow (\forall c{:}sp(s, c\emptyset) \Rightarrow sp(s,s.X))]$

$\Rightarrow$ { Identity 2.2.4 }

$[c\emptyset \wedge s.X \Rightarrow (\forall c{:}sp(s, c\emptyset) \Rightarrow X)]$

Theorem 2.3.2: Context Sensitive Refinement

$\{P\}s\{?Q\} \leq t \quad = \quad [P \wedge s.Q \Rightarrow t.Q]$

----

first, the left to right direction:

$\{P\}s\{?Q\} \leq t$

= { def of refinement }

$\forall X{:}[P \wedge s.Q \wedge [Q \Rightarrow X] \Rightarrow P \wedge t.X]$

= { simplify }

$\forall X{:}[P \wedge s.Q \wedge [Q \Rightarrow X] \Rightarrow t.X]$

$\Rightarrow$ { Q for X }

$[P \wedge s.Q \Rightarrow t.Q]$

Now, the right to left direction:

$\{P\}s\{?Q\} \leq t$

= { def of refinement }

$\forall X{:}[P \wedge s.Q \wedge [Q \Rightarrow X] \Rightarrow P \wedge t.X]$

= { simplify }

$\forall X{:}[P \wedge s.Q \wedge [Q \Rightarrow X] \Rightarrow t.X]$

= { monotonicity }

$\forall X{:}[P \wedge s.Q \wedge [Q \Rightarrow X] \wedge [t.Q \Rightarrow t.X] \Rightarrow t.X]$

$\Leftarrow$

$[P \wedge s.Q \Rightarrow t.Q]$

Theorem 2.5.2: Law of monotonicity for s*:
$[P \Rightarrow Q] \Rightarrow [s^*.P \Rightarrow s^*.Q]$

$[P \Rightarrow Q]$

$=$

$[\neg P \Leftarrow \neg Q]$

$\Rightarrow$ { assuming monotonicity of s }

$[s.\neg P \Leftarrow s.\neg Q]$

$=$

$[\neg s.\neg P \Rightarrow \neg s.\neg Q]$

$\Rightarrow$

$[(s.T) \wedge (\neg s.\neg P) \Rightarrow (s.T) \wedge (\neg s.\neg Q)]$

$=$

$[s^*.P \Rightarrow s^*.Q]$

Theorem 2.5.3: Conjunctive law for s*, in one direction:
$[s^*.P \wedge Q \Rightarrow s^*.P \wedge s^*.Q]$

$wp(s^*,P \wedge Q)$

$=$

$wp(s,T) \wedge \neg wp(s,\neg P \vee \neg Q)$

$\Rightarrow$ { assuming wp() obeys disjunctive law }

$wp(s,T) \wedge \neg wp(s,\neg P) \wedge \neg wp(s,\neg Q))$

$=$

$wp(s^*,P) \wedge wp(s^*,Q)$

Theorem 2.5.4: Disjunctive law for s*, in both directions:
$[s^*.(P \vee Q) = s^*.P \vee s^*.Q]$

$wp(s^*, P \vee Q)$

$=$

$wp(s,T) \wedge \neg wp(s,\neg P \wedge \neg Q)$

$=$

$wp(s,T) \wedge \neg(wp(s,\neg P) \wedge wp(s,\neg Q))$

$=$

$wp(s,T) \wedge (\neg wp(s,\neg P) \vee \neg wp(s,\neg Q))$

$=$

$wp(s,T) \wedge \neg wp(s,\neg P) \vee wp(s,T) \wedge \neg wp(s,\neg Q)$

$=$

$wp(s^*, P) \vee wp(s^*, Q)$

Lemma 4.1.1.1:

$|[\ \text{TypeB b; s}\ ]| \le |[\ \text{TypeB b; TypeK k; b*:I ; s}\ ]|$

if $[\exists b{:}I]$

$|[\ \text{TypeB b; s}\ ]| \le |[\ \text{TypeB b; TypeK k; b*:I ; s}\ ]|$

= { for X a predicate in o, b }

$\forall X{:}[(\forall b{:}s.X) \Rightarrow (\forall b,k{:}(\exists b{:}I \wedge s.X))]$

= { b not free in rhs, absorption of k }

$\forall X{:}[(\forall b{:}s.X) \Rightarrow (\exists b{:}I \wedge s.X)]$

= { calculus }

$\forall X{:}[(\forall b{:}s.X) \Rightarrow (\exists b{:}I) \wedge (\forall b{:}s.X)]$

= { calculus }

$\forall X{:}[(\forall b{:}s.X) \Rightarrow (\exists b{:}I)]$

= { T for X }

$[(\forall b{:}s.T) \Rightarrow (\exists b{:}I)]$

$\Leftarrow$

$[\exists b{:}I]$

= { in words }

for all values of k and common variables, there exists a b which makes I true.

Lemma 4.1.1.2:

$|[\ \text{TypeB b; s}\ ]| \le |[\ \text{TypeB b; TypeK k; k:}(\exists b{:}I)\text{; b*:I; s}\ ]|$

$|[\ \text{TypeB b; s}\ ]| \le |[\ \text{TypeB b; TypeK k; k:}(\exists b{:}I)\text{; b*:I; s}\ ]|$

=

$\forall X{:}[(\forall b{:}s.X) \Rightarrow (\forall b,k{:}(\forall k{:}(\exists b{:}I) \Rightarrow (\exists b{:}I \wedge s.X)))]$

= { absorption of k,b }

$\forall X{:}[(\forall b{:}s.X) \Rightarrow (\forall k{:}(\exists b{:}I) \Rightarrow (\exists b{:}I \wedge s.X))]$

= { absorption of k, importation }

$\forall X{:}[(\forall b{:}s.X) \wedge (\exists b{:}I) \Rightarrow (\exists b{:}I \wedge s.X)]$

= { calculus }

TRUE

Lemma 4.1.2.1:
$s \leq |[\text{TypeK } k;\ k{:}I;\ b^{*}{:}I;\ s\ ]|$

$s \leq |[\text{TypeK } k;\ k{:}I;\ b^{*}{:}I;\ s\ ]|$

= { def of Refinement }

$\forall X{:}[s.X \Rightarrow (\forall k{:}I \Rightarrow (\exists b{:}I \wedge s.X))]$

= { absorption, importation }

$\forall X{:}[I \wedge s.X \Rightarrow (\exists b{:}I \wedge s.X))]$

=

TRUE

Theorem 4.1.2.2:
$\{P\}b^{*}{:}I \leq \{P\}b{:}I$ if $[P \Rightarrow (\exists! b{:}I)]$

$\{P\}b^{*}{:}I \leq \{P\}b{:}I$

= { def of refinement }

$\forall X{:}[P \wedge (\exists b{:}I \wedge X) \Rightarrow P \wedge (\exists b{:}I) \wedge (\forall b{:}I \Rightarrow X)]$

= { P and (∃b:I) in rhs implied by lhs}

$\forall X{:}[P \wedge (\exists b{:}I \wedge X) \Rightarrow (\forall b{:}I \Rightarrow X)]$

= { b no free in rhs }

$\forall X{:}[(\exists b{:}P) \wedge (\exists b{:}I \wedge X) \Rightarrow (\forall b{:}I \Rightarrow X)]$

= { if $[P \Rightarrow (\exists! b{:}I)]$ }

$\forall X{:}[(\exists b{:}P) \wedge (\exists! b{:}I) \wedge (\exists b{:}I \wedge X) \Rightarrow (\forall b{:}I \Rightarrow X)]$

= { absorption of b, importation of I }

$\forall X{:}[(\exists b{:}P) \wedge (\exists! b{:}I) \wedge (\exists! b{:}I \wedge X) \wedge I \Rightarrow X)]$

= { $I \wedge (\exists! b{:}I) \wedge (\exists! b{:}I \wedge X) \Rightarrow I \wedge X)]$

TRUE

Theorem 4.1.2.3:
$\{P\}b^{*}{:}I\{?Q\} \leq \{P\}b{:}Q\{?Q\}$ always

$\{P\}b^{*}{:}I;\{?Q\} \leq \{P\}b{:}Q;\{?Q\}$

= { Theorem 2.3.2 }

$P \wedge (\exists b{:}I \wedge Q) \Rightarrow P \wedge (\exists b{:}Q) \wedge (\forall b{:}Q \Rightarrow Q)$

=

TRUE

Theorem 4.4.5: Deriving Z Procedural Refinement

$OP1 \le OP2$

=

$[\mathit{pre}OP1' \Rightarrow \mathit{pre}OP2'] \wedge$

$[\mathit{pre}OP1' \wedge OP2' \Rightarrow OP1']$

We show implication in both directions:

$OP1 \le OP2$

= { refinement rule }

$\forall X:[wps(OP1, X) \Rightarrow wps(OP2, X)]$

$\Rightarrow$ { c\$bef.OP1.c for X, call it P }

$[wps(OP1, P) \Rightarrow wps(OP2, P)]$

= { def of wps() }

$[(\exists c':OP1') \wedge (\forall c':OP1' \Rightarrow P`) \Rightarrow$

$(\exists c':OP2') \wedge (\forall c':OP2' \Rightarrow P`)]$

$\Rightarrow$ { add to left side }

$[(c\$bef = c) \wedge (\exists c':OP1') \wedge (\forall c':OP1' \Rightarrow P`) \Rightarrow$

$(\exists c':OP2') \wedge (\forall c':OP2' \Rightarrow P`)]$

= { simplify $(c\$bef=c) \wedge P`$ to c.OP1.c' }

$[(\exists c':OP1') \wedge (\forall c':OP1' \Rightarrow OP1') \Rightarrow$

$(\exists c':OP2') \wedge (\forall c':OP2' \Rightarrow OP1')]$

= { simplify }

$[(\exists c':OP1') \Rightarrow$

$(\exists c':OP2') \wedge (\forall c':OP2' \Rightarrow OP1')]$

= { split and simplify }

$[(\exists c':OP1') \Rightarrow (\exists c':OP2')] \wedge$

$[(\exists c':OP1') \Rightarrow (\forall c':OP2' \Rightarrow OP1')]$

= { importation }

$[\mathit{pre}OP1' \Rightarrow \mathit{pre}OP2'] \wedge$

$[\mathit{pre}OP1' \wedge OP2' \Rightarrow OP1']$

Now we show implication in the other direction:

$[pre\mathrm{OP1}' \Rightarrow pre\mathrm{OP2}'] \;\wedge$
$[pre\mathrm{OP1}' \wedge \mathrm{OP2}' \Rightarrow \mathrm{OP1}']$
= { exportation, unsplit }
$[(\exists c':\mathrm{OP1}') \Rightarrow$
$(\exists c':\mathrm{OP2}') \wedge (\forall c':\mathrm{OP2}' \Rightarrow \mathrm{OP1}')]$
$\Rightarrow$ { add the same thing to both sides }
$\forall X:[(\exists c':\mathrm{OP1}') \wedge (\forall c':\mathrm{OP1}' \Rightarrow X`) \Rightarrow$
$(\exists c':\mathrm{OP2}') \wedge (\forall c':\mathrm{OP2}' \Rightarrow \mathrm{OP1}') \wedge (\forall c':\mathrm{OP1}' \Rightarrow X`)]$
$\Rightarrow$ { weaken the right side }
$\forall X:[(\exists c':\mathrm{OP1}') \wedge (\forall c':\mathrm{OP1}' \Rightarrow X`) \Rightarrow$
$(\exists c':\mathrm{OP2}') \wedge (\forall c':\mathrm{OP2}' \Rightarrow X`)\ ]$
=.{ def of wps() }
$\forall X:[\mathrm{wps}(\mathrm{OP1}, X) \Rightarrow \mathrm{wps}(\mathrm{OP2}, X)]$
= { refinement rule }
$\mathrm{OP1} \leq \mathrm{OP2}$

Theorems 4.4.6, 4.4.8, 4.4.9, 4.4.10: exercise.

Lemmas 4.4.1.3 and 4.4.1.4: exercise.
(hint: plug into Definition 4.4.7 using Theorem 4.4.10.)

Theorem 4.4.1.5: Deriving Z Data Refinement

$(II^* \; ; \; S) \leq (T \; ; \; II^*)$

$=$

$[I \wedge \mathit{pre}S' \Rightarrow \mathit{pre}T'] \;\wedge$

$[I \wedge \mathit{pre}S' \wedge T' \Rightarrow (\exists b'{:}I' \wedge S')]$

The major challenge in this proof is managing the myriad renamings. Toward this end we rename the various predicates at the start of the proof. We partition the program variables into abstract variables b and all other variables o (common and concrete).

Assume II3 = b''.o''.II3.bo is the first II,
S = bo.S.b'o', T2 = o.T.o'', T = o.T.o',
II2 = b''o''.II2.b'o' is the second II, X = b\`o\`.X.b'o',
II3 is $(I \wedge (o'' = o))$, and II2 is $(I' \wedge (o''=o'))$

$II3^*;S \leq (T2;II2^*)$

= { def of refinement }

$\forall X{:}[wps(II3^*,\; wps(S,\; X)) \Rightarrow\; wps(T2,\; wps(II2^*,\; X))]$

= {def of wps }

$\forall X{:}[wps(II3^*,\; (\exists b'o'{:}S) \wedge (\forall b'o'{:}S \Rightarrow X)) \Rightarrow$

$wps(T2,\; (\exists b'o'{:}II2 \wedge X))]$

= {def of wps }

$\forall X{:}[(\exists bo{:}II3 \wedge (\exists b'o'{:}S) \wedge (\forall b'o'{:}S \Rightarrow X)) \Rightarrow$

$(\exists o''{:}T2) \wedge (\forall o''{:}T2 \Rightarrow (\exists b'o'{:}II2 \wedge X))]$

= { 1-point from II2 and II3 }

$\forall X{:}[(\exists b{:}I[o''/o] \wedge (\exists b'o'{:}S[o''/o]) \wedge (\forall b'o'{:}S[o''/o] \Rightarrow X)) \Rightarrow$

$(\exists o''{:}T2) \wedge (\forall o''{:}T2 \Rightarrow (\exists b'{:}I'[o''/o'] \wedge X[o''/o']))]$

= { rename free o'' to o, bound o'' to o' }

$\forall X{:}[(\exists b{:}I \wedge (\exists b'o'{:}S) \wedge (\forall b'o'{:}S \Rightarrow X)) \Rightarrow$

$(\exists o'{:}T) \wedge (\forall o'{:}T \Rightarrow (\exists b'{:}I' \wedge X))]$

= { Lemma 4.4.1.7 }

$[I \wedge \mathit{pre}S \Rightarrow \mathit{pre}T \wedge (\forall o'{:}T \Rightarrow (\exists b'{:}I' \wedge S))]$

= {split }

$[I \wedge \mathit{pre}S \Rightarrow \mathit{pre}T\ ]$

$[I \wedge \mathit{pre}S \Rightarrow (\forall o'{:}T \Rightarrow (\exists b'{:}I' \wedge S))]$

= { absorption, importation }

$[I \wedge \mathit{pre}S \Rightarrow \mathit{pre}T\ ]$

$[I \wedge \mathit{pre}S \wedge T \Rightarrow (\exists b'{:}I' \wedge S))]$

$=$

Z data refinement rule

Theorem 4.4.1.6: exercise.

Lemma 4.4.1.7:

Assume $S = bo.S.b'o'$, $T = o.T.o'$, $X = b`o`.X.b'o'$, then:

$\forall X:[(\exists b:I \wedge (\exists b'o':S) \wedge (\forall b'o':S \Rightarrow X)) \Rightarrow$
$\quad (\exists o':T) \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge X))]$

=

$[I \wedge preS \Rightarrow preT \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge S))]$

We prove implication in both directions:

$[I \wedge preS \Rightarrow preT \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge S))]$

$\Rightarrow$ { add the same thing to both sides }

$\forall X:[I \wedge preS \wedge (\forall b'o':S \Rightarrow X) \Rightarrow$
$\quad preT \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge S)) \wedge (\forall b'o':S \Rightarrow X)]$

$\Rightarrow$ { transitivity of implication, weaken right side }

$\forall X:[I \wedge preS \wedge (\forall b'o':S \Rightarrow X) \Rightarrow$
$\quad preT \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge X))]$

= { absorption }

$\forall X:[(\exists b:I \wedge preS \wedge (\forall b'o':S \Rightarrow X)) \Rightarrow$
$\quad preT \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge X))]$

Now the other direction:

$\forall X:[(\exists b:I \wedge \mathit{pre}S \wedge (\forall b'o':S \Rightarrow X)) \Rightarrow$
$\quad \mathit{pre}T \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge X))]$

= { absorption }

$\forall X:[I \wedge \mathit{pre}S \wedge (\forall b'o':S \Rightarrow X) \Rightarrow$
$\quad \mathit{pre}T \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge X))]$

$\Rightarrow$ { $S2 = b`o`.S.b'o'$ for X}

$[I \wedge \mathit{pre}S \wedge (\forall b'o':S \Rightarrow S2) \Rightarrow$
$\quad \mathit{pre}T \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge S2))]$

$\Rightarrow$ { add (o=o') and (b=b') to left side, apply 1-point }

$[I \wedge \mathit{pre}S \wedge (\forall b'o':S \Rightarrow S) \Rightarrow$
$\quad \mathit{pre}T \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge S))]$

= { simplify }

$[I \wedge \mathit{pre}S \Rightarrow \mathit{pre}T \wedge (\forall o':T \Rightarrow (\exists b':I' \wedge S))]$

Theorem 4.5.2: exercise.

Theorem 4.6.1: see the proof of Theorem 5.1.1.

Theorem 5.1:

$s \le t \;=\; [c\emptyset \wedge s.T \Rightarrow t.sp(s, c\emptyset)]$

----

first we prove it from left to right:

$s \le t$

= { def refinement }

$\forall X:[s.X \Rightarrow t.X]$

$\Rightarrow$ { sp(s, cø) for X }

$[s.sp(s, c\emptyset) \Rightarrow t.sp(s, c\emptyset)]$

$\Rightarrow$ { Identity 2.2.3 }

$[c\emptyset \wedge s.T \Rightarrow t.sp(s, c\emptyset)]$

Now, right to left:

$[s.X \Rightarrow t.X]$ for arbitrary X

= { extend quantification }

$[c\emptyset \wedge s.X \Rightarrow t.X]$

= { $c\emptyset \wedge s.X = c\emptyset \wedge s.T \wedge s.X$ }

$[c\emptyset \wedge s.T \wedge s.X \Rightarrow t.X]$

$\Leftarrow$ { Identity 2.2.5 }

$[c\emptyset \wedge s.T \wedge (\forall c:sp(s, c\emptyset) \Rightarrow X) \Rightarrow t.X]$

$\Leftarrow$ { monotonicity of t }

$[c\emptyset \wedge s.T \wedge (\forall c:t.sp(s, c\emptyset) \Rightarrow t.X) \Rightarrow t.X]$

$\Leftarrow$

$[c\emptyset \wedge s.T \Rightarrow t.sp(s, c\emptyset)]$

Theorem 5.1.1: Formula for $\downarrow$REFINEMENT

$s;\ k:I2;\ \{b?TRUE\} \le k:I1;\ t;\ \{b?TRUE\}$

=

$b^*:I1;\ s\ \{k?TRUE\} \le t;\ b^*:I2;\ \{k?TRUE\}$

=

$[c\emptyset \wedge I1 \wedge s.T \Rightarrow t.(\exists b:I2 \wedge sp(s, c\emptyset))]$

Lemmas 5.1.7 and 5.1.8

Theorem 5.1.2: Formula for $\uparrow$REFINEMENT

$b{:}I1;\ s;\ \{k?TRUE\} \le t;\ b{:}I2;\ \{k?TRUE\}$

=

$[k\emptyset \wedge o\emptyset \wedge (\forall b{:}I1 \Rightarrow s.T) \Rightarrow t.(\forall b{:}I2 \Rightarrow sp(s,\ I1[k\emptyset/k] \wedge o\emptyset))]$

$b{:}I1;\ \{k?TRUE\};\ s;\ \le t;\ b{:}I2;\ \{k?TRUE\}$

= { theorem 5.1 }

$[b\emptyset \wedge k\emptyset \wedge o\emptyset \wedge (\forall b{:}I1 \Rightarrow s.T) \Rightarrow$

$t.(\forall b{:}I2 \Rightarrow (\forall k{:}sp(s,\ o\emptyset \wedge (\exists k{:}k\emptyset \wedge I1))))]$

= { 1-point rule }

$[b\emptyset \wedge k\emptyset \wedge o\emptyset \wedge (\forall b{:}I1 \Rightarrow s.T) \Rightarrow$

$t.(\forall b{:}I2 \Rightarrow (\forall k{:}sp(s,\ o\emptyset \wedge I1[k\emptyset/k])))]$

= { sp(s,..) not free in k, rhs not free in bø, bø only free in bø }

$[k\emptyset \wedge o\emptyset \wedge (\forall b{:}I1 \Rightarrow s.T) \Rightarrow t.(\forall b{:}I2 \Rightarrow sp(s,\ I1[k\emptyset/k] \wedge o\emptyset))]$

Theorem 5.1.3:

$s';\{?Q\}\ \ll_{\mathbf{DN}}\ t$

=

$[I1 \wedge s'.Q \Rightarrow t.(\exists b{:}I2 \wedge Q)]$

$s';\{?Q\}\ \ll_{\mathbf{DN}}\ t$

= { def of $\ll_{\mathbf{DN}}$ }

$b^*{:}I1;\ s';\{?Q\};\ \{k?T\} \le t;\ b^*{:}I2;\ \{k?T\}$

= { refinement }

$\forall X{:}[(\exists b{:}I1 \wedge s'.Q) \wedge [Q \Rightarrow (\forall k{:}X)] \Rightarrow t.(\exists b{:}I2 \wedge (\forall k{:}X))]$

= { Q for X }

$[(\exists b{:}I1 \wedge s'.Q) \Rightarrow t.(\exists b{:}I2 \wedge Q)]$

= { rhs not free in b }

$[I1 \wedge s'.Q \Rightarrow t.(\exists b{:}I2 \wedge Q)]$

Theorem 5.1.4:

$s';\{?Q\}\ \ll_{\mathbf{UP}}\ t$

=

$[(\forall b{:}I1 \Rightarrow s'.Q) \Rightarrow t.(\forall b{:}I2 \Rightarrow Q)]$

$s';\{?Q\}\ \ll_{\mathbf{UP}}\ t$

= { def of $\ll_{\mathbf{UP}}$ }

$b{:}I1;\ \{k?TRUE\};\ s';\{?Q\} \le t;\ b{:}I2;\ \{k?TRUE\};$

= { Theorem 5.1 }

$[(\forall b{:}I1 \Rightarrow (\forall k{:}s'.Q)) \Rightarrow t.(\forall b{:}I2 \Rightarrow (\forall k{:}Q))]$

= { Q and s'.Q do not contain k, }

$[(\forall b{:}I1 \Rightarrow s'.Q) \Rightarrow t.(\forall b{:}I2 \Rightarrow Q)]$

Theorem 5.1.5:
if s1 $\ll_{DN}$ t1 with invariants I1 and I2, and
s2 $\ll_{DN}$ t2 with invariants I1' and I2', then
s1;s2 $\ll_{DN}$ t1;t2 with invariants I1 and I2' if
$[I2 \Rightarrow I1']$.

---

given:

s1; k:I2; {b?TRUE} ≤ k:I1; t1; {b?TRUE}

s2; k:I2'; {b?TRUE} ≤ k:I1'; t2; {b?TRUE}

prove:

s1;s2; k:I2'; {b?TRUE} ≤ k:I1; t1;t2; {b?TRUE}

if $[I2 \Rightarrow I1']$

We refine the lhs to the rhs in steps:

s1;s2; k:I2'; {b?TRUE}

≤ { using refinement of s2 }

s1; k:I1'; t2; {b?TRUE}

= { t2; {b?TRUE} ≤ {b?TRUE}; t2; {b?TRUE} }

s1; k:I1'; {b?TRUE}; t2;

≤ { (k:I1' ≤ k:I2) = $[I2 \Rightarrow I1']$ }

s1; k:I2; {b?TRUE}; t2; {b?TRUE}

≤ { using refinement of s1 }

k:I1; t1; {b?TRUE} t2; {b?TRUE}

≤ { {b?P} refines to skip }

k:I1; t1; t2; {b?TRUE}

Theorem 5.1.6:
similar to 5.1.5.

Lemma 5.1.7:

b*:I1; s {k?TRUE} ≤ t; b*:I2; {k?TRUE}

=

$[I1 \wedge c\emptyset \wedge s.T \Rightarrow t.(\exists b{:}I2 \wedge sp(s,\ c\emptyset))]$

We cannot use theorem 5.1 for this because there is no formula for sp(b*:I, X).

b*:I1; s {k?TRUE} ≤ t; b*:I2; {k?TRUE}

=

$\forall X{:}[(\exists b{:}I1 \wedge s.(\forall k{:}X)) \Rightarrow t.(\exists b{:}I2 \wedge (\forall k{:}X))]$

=

$\forall X{:}[(\exists b{:}I1 \wedge s.X) \Rightarrow t.(\exists b{:}I2 \wedge X)]$ for X not containing k free.

First we prove the left to right direction:
by plugging in sp(s, cø) for X we get

$[I1 \wedge s.sp(s, cø) \Rightarrow t.(\exists b{:}I2 \wedge sp(s, cø))]$

$\Rightarrow$ { Identity 2.2.3 }

$[I1 \wedge cø \wedge s.T \Rightarrow t.(\exists b{:}I2 \wedge sp(s, cø))]$

Now we prove the left to right direction:

$[I1 \wedge cø \wedge s.T \Rightarrow t.(\exists b{:}I2 \wedge sp(s, cø))] \Rightarrow \forall X{:}[I1 \wedge s.X \Rightarrow t.(\exists b{:}I2 \wedge X)]$

= { take $\forall X$: outside }

$\forall X{:}|[I1 \wedge cø \wedge s.T \Rightarrow t.(\exists b{:}I2 \wedge sp(s, cø))] \Rightarrow [I1 \wedge s.X \Rightarrow t.(\exists b{:}I2 \wedge X)]|$

= { absorption, importation }

$\forall X{:}|[I1 \wedge cø \wedge s.T \Rightarrow t.(\exists b{:}I2 \wedge sp(s, cø))] \wedge I1 \wedge s.X \Rightarrow t.(\exists b{:}I2 \wedge X)]$

$\Leftarrow$ { for arbitrary X over b,o }

$|[I1 \wedge cø \wedge s.T \Rightarrow t.(\exists b{:}I2 \wedge sp(s, cø))] \wedge I1 \wedge s.X \Rightarrow t.(\exists b{:}I2 \wedge X)]$

$\Leftarrow$ { $I1 \wedge cø \wedge s.X \Rightarrow I1 \wedge cø \wedge s.T$ }

$[t.(\exists b{:}I2 \wedge sp(s, cø)) \wedge cø \wedge s.X \Rightarrow t.(\exists b{:}I2 \wedge X)]$

$\Leftarrow$ { Identity 2.2.5 }

$[t.(\exists b{:}I2 \wedge sp(s, cø)) \wedge (\forall c{:}sp(s, cø) \Rightarrow X) \Rightarrow t.(\exists b{:}I2 \wedge X)]$

$\Leftarrow$ { $(\forall c{:}sp(s, cø) \Rightarrow X) \Rightarrow (\forall c{:}I \wedge sp(s, cø) \Rightarrow I \wedge X)$ }

$[t.(\exists b{:}I2 \wedge sp(s, cø)) \wedge (\forall c{:}I2 \wedge sp(s, cø) \Rightarrow I2 \wedge X) \Rightarrow t.(\exists b{:}I2 \wedge X)]$

$\Leftarrow$ { $[(\forall c{:}X \Rightarrow Y) \Rightarrow (\forall c{:}(\exists b{:}X) \Rightarrow (\exists b{:}Y))]$ }

$[t.(\exists b{:}I2 \wedge sp(s, cø)) \wedge (\forall c{:}(\exists b{:}I2 \wedge sp(s, cø)) \Rightarrow (\exists b{:}I2 \wedge X)) \Rightarrow$
$t.(\exists b{:}I2 \wedge X)]$

$\Leftarrow$ { monotonicity of t }

$[t.(\exists b{:}I2 \wedge sp(s, cø)) \wedge (\forall c{:}t.(\exists b{:}I2 \wedge sp(s, cø)) \Rightarrow t.(\exists b{:}I2 \wedge X)) \Rightarrow$
$t.(\exists b{:}I2 \wedge X)]$

=

TRUE

Lemma 5.1.8:

s; k:I2; {b?TRUE} $\leq$ k:I1; t; {b?TRUE}

=

$[cø \wedge I1 \wedge s.T \Rightarrow t.(\exists b{:}I2 \wedge sp(s, cø))]$

where cø is $(b=bø) \wedge (o=oø)$

----

s; k:I2; {b?TRUE} $\leq$ k:I1; {b?TRUE} t;

= { using Theorem 5.1 }

$[cø \wedge kø \wedge s.T \Rightarrow (\forall k{:}I1 \Rightarrow t.(\exists b{:}I2 \wedge (\exists k{:}sp(s,cø \wedge kø))))]$

= { $sp(s, cø \wedge kø) = kø \wedge sp(s, cø)$, sp(s, cø) not free in k }

$[cø \wedge s.T \wedge I1 \Rightarrow t.(\exists b{:}I2 \wedge sp(s, cø))]$

Theorem 5.2.1: Nonlocal Data Refinement using ↓REFINEMENT:

$s \le |[\ \mathrm{TypeK}\ k;\ k{:}I1;\ t;\ \{\exists b{:}I2 \wedge sp(s, c\emptyset)\}\ b{:}P\ ]|$

if $[c\emptyset \wedge s.T \Rightarrow s.(\forall k{:}I2 \Rightarrow (\forall b{:}P \Rightarrow sp(s, c\emptyset)))]$ and

$s\ \ll\mathbf{DN}\ t$

----

s

$\le$ { if $[c\emptyset \wedge s.T \Rightarrow s.(\forall k{:}I2 \Rightarrow (\forall b{:}P \Rightarrow sp(s, c\emptyset)))]$, from Theorem 5.1 }

$|[\ \mathrm{TypeK}\ k;\ s;\ k{:}I2;\ b{:}P\ ]|$

= { can add $\{b?TRUE\}$ before $b{:}P$ }

$|[\ \mathrm{TypeK}\ k;\ s;\ k{:}I2;\ \{b?TRUE\}\ b{:}P\ ]|$

$\le$ { ↓REFINEMENT }

$|[\ \mathrm{TypeK}\ k;\ k{:}I1;\ t;\ \{\exists b{:}I2 \wedge sp(s, c\emptyset)\}\ b{:}P\ ]|$

Corollary 5.2.4: Special Case of $[(\exists b{:}I2 \wedge Q) \Rightarrow (\exists! b{:}I2 \wedge Q)]$:

$s\{Q\} \le |[\ \mathrm{TypeK}\ k;\ k{:}I1;\ t;\ \{\exists b{:}I2 \wedge Q\}\ b{:}(I2 \wedge Q)\ ]|$

if $s\{Q\}\ \ll\mathbf{DN}\ t$

We use the statement none:P which essentially is a promise to add an assertion of {P} at this point in the program, sometime in the future. It can be used to create the miracle statement in [Morris, 89] and [Morgan, 87b].

$s\{Q\}$

$\le$ { Lemma 5.2.10 }

$|[\ \mathrm{TypeK}\ k;\ s\{Q\};\ k{:}I2;\ none{:}(\exists b{:}I2 \wedge Q \wedge sp(s, c\emptyset));\ b{:}(I2 \wedge Q)\ ]|$

= { add $\{b?TRUE\}$ before $b{:}P$ }

$|[\ \mathrm{TypeK}\ k;\ s\{Q\};\ k{:}I2;\ \{b?TRUE\}\ none{:}(\exists b{:}I2 \wedge sp(s, c\emptyset));\ b{:}(I2 \wedge Q)\ ]|$

$\le$ { $s\{Q\}\ \ll\mathbf{DN}\ t$ }

$|[\ \mathrm{TypeK}\ k;\ k{:}I1;\ t;\ \{\exists b{:}I2 \wedge Q \wedge sp(s, c\emptyset)\};\ none{:}(\exists b{:}I2 \wedge sp(s, c\emptyset));\ b{:}(I2 \wedge Q)$
$]|$

= { $\{P\}none{:}P$ refines to $\{P\}$ }

$|[\ \mathrm{TypeK}\ k;\ k{:}I1;\ t;\ \{\exists b{:}I2 \wedge Q \wedge sp(s, c\emptyset)\}\ b{:}(I2 \wedge Q)\ ]|$

Theorem 5.2.5: Nonlocal Data Refinement Using $\uparrow$REFINEMENT:

s ≤ |[TypeK k; k:P; t; b:I2]|

if [$c\emptyset \wedge$ s.T $\wedge$ P $\Rightarrow$ ($\forall$b:I1 $\Rightarrow$ s.sp(s, $c\emptyset$))] and

s **‹‹UP** t

----

s

≤ { if [$c\emptyset \wedge$ s.T $\wedge$ P $\Rightarrow$ ($\forall$b:I1 $\Rightarrow$ s.sp(s, $c\emptyset$))] from Thm. 5.1 }

|[TypeK k; k:P; b:I1; {k?TRUE} s;]|

≤ { $\uparrow$REFINEMENT }

|[TypeK k; k:P; t; b:I2]|

Corollary 5.2.7: Special Case s is of the form {b?Q}s',

P is ($\exists$b:I1' $\wedge$ Q), I1 is I1' $\wedge$ Q

{b?Q}s' ≤ |[TypeK k; k:($\exists$b:Q $\wedge$ I1'); t; b:I2]|

if {b?Q}s **‹‹UP** t with I1 = I1' $\wedge$ Q

{b?Q}s

≤ { Lemma 5.2.11 }

|[TypeK k; k:P; b:I1' $\wedge$ Q; {b?Q}s; {k?TRUE}]|

≤ { $\uparrow$REFINEMENT }

|[TypeK k; k:P; t; b:I2]|

= { substitution for P }

|[TypeK k; k:($\exists$b:I1' $\wedge$ Q); t; b:I2]|

Theorem 5.2.8:

{b,c?P} = {b?P}{b,c?P}

Exercise.

Corollary 5.2.9: Special Case if $[(\exists b{:}I1' \wedge Q) \Rightarrow (\exists! b{:}I1' \wedge Q)]$

$\{Q\}s \leq |[\ TypeK\ k;\ k{:}I1';\ t;\ b{:}I2\ ]|$

if $\{Q\}s \ll_{\mathbf{UP}} t$ with $I1 = I1' \wedge Q$

$$\{Q\}s \leq |[\ TypeK\ k;\ k{:}I1';\ b{:}I1' \wedge Q;\ \{Q\}s;\ ]|$$

= { def of refinement }

$$[c\emptyset \wedge Q \wedge s.T \wedge I1' \Rightarrow (\forall b{:}I1' \wedge Q \Rightarrow Q \wedge s.sp(s,\ c\emptyset))]$$

= { Identity 2.2.3 with $P=c\emptyset$ }

$$[c\emptyset \wedge Q \wedge s.T \wedge I1' \wedge s.sp(s,\ c\emptyset) \Rightarrow (\forall b{:}I1' \wedge Q \Rightarrow s.sp(s,\ c\emptyset))]$$

= { absorption }

$$[(\exists b{:}c\emptyset \wedge Q \wedge s.T \wedge I1' \wedge s.sp(s,\ c\emptyset)) \Rightarrow (\forall b{:}I1' \wedge Q \Rightarrow s.sp(s,\ c\emptyset))]$$

= { if $[(\exists b{:}I1' \wedge Q) \Rightarrow (\exists! b{:}I1' \wedge Q)]$ }

TRUE

Lemma 5.2.10: for Corollary 5.2.4

$s\{Q\} \leq |[\ TypeK\ k;\ s\{Q\};\ k{:}I2;\ none{:}(\exists b{:}I2 \wedge Q \wedge sp(s,\ c\emptyset));\ b{:}(I2 \wedge Q)\ ]|$

if $[(\exists b{:}I2 \wedge Q) \Rightarrow (\exists! b{:}I2 \wedge Q)]$

$$s\{Q\} \leq |[\ TypeK\ k;\ s\{Q\};\ k{:}I2;\ none{:}(\exists b{:}I2 \wedge Q \wedge sp(s,\ c\emptyset));\ b{:}(I2 \wedge Q)\ ]|$$

= { theorem 5.1 }

$$[c\emptyset \wedge s.Q \Rightarrow s.(Q \wedge (\forall k{:}I2 \wedge (\exists b{:}I2 \wedge Q \wedge sp(s,\ c\emptyset)) \Rightarrow (\forall b{:}I2 \wedge Q \Rightarrow Q \wedge sp(s,\ c\emptyset))))]$$

= { if $[(\exists b{:}I2 \wedge Q) \Rightarrow (\exists! b{:}I2 \wedge Q)]$ }

$$[c\emptyset \wedge s.Q \Rightarrow s.Q]$$

=

TRUE

Lemma 5.2.11: for Corollary 5.2.7

if $\{b?Q\}s' = s$, $(\exists b{:}Q \wedge I1') = P$, $I1' \wedge Q = I1$, then

$s \le$ |[TypeK k; k:P; b:I1; s; {k?TRUE}]|

$[c\emptyset \wedge s.T \wedge P \Rightarrow (\forall b{:}I1 \Rightarrow s.sp(s, c\emptyset))]$

= { {b?Q}s' for s, (∃b:Q ∧ I1') for P, I1' ∧ Q for I1 }

$[c\emptyset \wedge (\forall b{:}Q \Rightarrow s'.T) \wedge (\exists b{:}Q \wedge I1') \Rightarrow$

$(\forall b{:}I1' \wedge Q \Rightarrow (\forall b{:}Q \Rightarrow s'.sp(s',Q \wedge (\exists b{:}c\emptyset))))]$

= { absorption, importation }

$[(\exists b{:}c\emptyset) \wedge (\forall b{:}Q \Rightarrow s'.T) \wedge (\exists b{:}Q \wedge I1') \Rightarrow$

$(\forall b{:}Q \wedge (\exists b{:}I1' \wedge Q) \Rightarrow s'.sp(s',Q \wedge (\exists b{:}c\emptyset)))]$

= { absorption, importation }

$[(\exists b{:}c\emptyset) \wedge (\forall b{:}Q \Rightarrow s'.T) \wedge (\exists b{:}Q \wedge I1') \wedge Q \Rightarrow s'.sp(s',Q \wedge (\exists b{:}c\emptyset))]$

= { adding }

$[(\exists b{:}c\emptyset) \wedge (\forall b{:}Q \Rightarrow s'.T) \wedge (\exists b{:}Q \wedge I1') \wedge Q \wedge s'.T \Rightarrow s'.sp(s',Q \wedge (\exists b{:}c\emptyset))]$

=

TRUE

Theorem 5.3.1: Local Data Refinement with ↓REFINEMENT

|[TypeB b; s]| ≤ |[TypeK k; k:(∃b:I1); t; {∃b:I2}]|

if s ‹‹**DN** t

|[TypeB b; s]|

= { Lemma 5.3.3 }

|[TypeB b; TypeK k; s; k:I2; {b?TRUE}]|

≤ { s ‹‹**DN** t }

|[TypeB b; TypeK k; k:I1; {b?TRUE} t;]|

≤ { Lemma 5.3.4 }

|[TypeB b; TypeK k; k:(∃b:I1); {b?TRUE} t;]|

≤ { b lines become virtual }

|[TypeK k; k:(∃b:I1); t;]|

Theorem 5.3.2: Local Data Refinement with ↑REFINEMENT

|[TypeB b; s]| ≤ |[TypeK k; k:(∃b:I1); t; {∃b:I2}]|

if s ‹‹**UP** t

|[TypeB b; s]|

= { Lemma 5.3.5 }

|[TypeB b; TypeK k; k:(∃b:I1); b:I1; {k?TRUE} s;]|

≤ { ↑REFINEMENT }

|[TypeB b; TypeK k; k:(∃b:I1); {b?TRUE} t;]|

≤ { remove virtual lines }

|[TypeK k; k:(∃b:I1); t;]|

Lemma 5.3.3:

|[TypeB b; s]| ≤ |[TypeB b; TypeK k; s; k:I2; {b?TRUE}]|

|[TypeB b; s]| ≤ |[TypeB b; TypeK k; s; k:I2; {b?TRUE}]|

= { def refinement }

$[o\emptyset \wedge (\forall b{:}s.T) \Rightarrow (\forall b{:}s.(\forall k{:}I2 \Rightarrow (\exists b{:}sp(s,o\emptyset))))]$

$\Leftarrow$

$[o\emptyset \wedge (\forall b{:}s.T) \Rightarrow (\forall b{:}s.sp(s,o\emptyset))]$

$\Leftarrow$ { $[X \Rightarrow Y] \Rightarrow [(\forall b{:}X) \Rightarrow (\forall b{:}Y)]$ for b on nonempty domain }

$[o\emptyset \wedge s.T \Rightarrow s.sp(s,o\emptyset))]$

= { Identity 2.2.3 }

TRUE

Lemma 5.3.4:

|[TypeB b; TypeK k; k:I1; {b?TRUE} t;]| ≤

|[TypeB b; TypeK k; k:(∃b:I1); {b?TRUE} t;]|

|[TypeB b; TypeK k; k:I1; {b?TRUE} t;]| ≤

|[TypeB b; TypeK k; k:(∃b:I1); {b?TRUE} t;]|

= { def refinement }

$\forall X{:}[(\forall b,k{:}(\forall k{:}I1 \Rightarrow \forall b{:}t.X)) \Rightarrow (\forall b,k{:}(\forall k{:}(\exists b{:}I1) \Rightarrow \forall b{:}t.X))]$

= { absorption }

$\forall X{:}[(\forall b,k{:}(\forall k{:}I1 \Rightarrow \forall b{:}t.X)) \Rightarrow (\forall k{:}(\exists b{:}I1) \Rightarrow \forall b{:}t.X)]$

= { absorption }

$\forall X{:}[(\forall b,k{:}I1 \Rightarrow \forall b{:}t.X) \Rightarrow (\forall k,b{:}I1 \Rightarrow \forall b{:}t.X)]$

=

TRUE

Lemma 5.3.5:

$|[$ TypeB b; s $]| \leq |[$ TypeB b; TypeK k; k:($\exists$b:I1); b:I1; {k?TRUE} s; $]|$

$|[$ TypeB b; s $]| \leq$

$|[$ TypeB b; TypeK k; k:($\exists$b:I1); b:I1; {k?TRUE} s; $]|$

= { def refinement }

$[o\emptyset \wedge (\forall b{:}s.T) \Rightarrow (\forall b,k{:}(\exists b{:}I2) \Rightarrow (\forall b{:}I2 \Rightarrow (\forall k{:}s.(\exists b{:}sp(s,\ o\emptyset))))))]$

=

$[o\emptyset \wedge (\forall b{:}s.T) \wedge (\exists b{:}I2) \Rightarrow (\forall b{:}I2 \Rightarrow (\forall k{:}s.(\exists b{:}sp(s,\ o\emptyset)))))]$

=

$[o\emptyset \wedge (\forall b{:}s.T) \wedge (\exists b{:}I2) \wedge I2 \Rightarrow (\forall k{:}s.(\exists b{:}sp(s,\ o\emptyset))))]$

=

$[o\emptyset \wedge (\forall b{:}s.T) \wedge (\exists b{:}I2) \wedge I2 \Rightarrow s.(\exists b{:}sp(s,\ o\emptyset))]$

=

$[o\emptyset \wedge s.T \wedge (\forall b{:}s.T) \wedge (\exists b{:}I2) \wedge I2 \Rightarrow s.(\exists b{:}sp(s,\ o\emptyset))]$

= { Identity 2.2.3 }

TRUE

Bibliography

Back, R.J.R.: A Calculus of Refinements for Program Derivation, Acta Informatica 25, 1988, pp. 593-624.

Bauer, F.L., Moller, B., Partsch, H., and Pepper, P.: Formal Program Construction by Transformation, IEEE Trans. on Soft. Eng., 15(2), February 1989, pp. 165-180.

Berry, D.M., Erlich, Z., and Lucena, C.J.: Pointers and Data Abstractions in High-Level Languages, Computer Languages vol. 2, 1977, pp. 135-148.

Dijkstra, E.W.: A Discipline of Programming, Englewood Cliffs, New Jersey. Prentice-Hall 1976.

Goguen, J.A., Thatcher, J.W., Wagner, E.G.: An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, in Current Trends in Programming Methodology, vol. 4, R. Yeh, ed. Prentice Hall, NJ., 1978, pp. 80-149.

Gries, D., and Prins, J.: A New Notion of Encapsulation, Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, SIGPLAN NOTICES 20(7), July 1985, pp. 131-139.

Guttag, J., Horowitz, E., Musser, D.: The Design of Data Type Specifications, in Current Trends in Programming Methodology, vol. 4, R. Yeh, ed. Prentice Hall, NJ., 1978, pp. 60-79.

Hayes, I., ed.: Specification Case Studies, Prentice Hall International, 1987.

He, J., Hoare, C.A.R., and Sanders, J.W.: Data Refinement Refined, ESOP 86, pp. 187-195.

Hehner, E.C.R., Gupta, L.E., Malton, A.J.: Predicative Methodology, Acta Informatica 23(5), 1986, pp. 487-505.

Hehner, E.C.R.: Predicative Programming, CACM 27(2), 1984, pp. 134-150.

Hoare, C.A.R.: Proof of Correctness of Data Representations, Acta Informatica 1, 1972, pp. 271-281.

Hoare, C.A.R., Hayes, I.J., et al.: Laws of Programming, CACM 30(8), August 1987a, pp. 672-686.

Hoare, C.A.R., He, J., and Sanders, J.W.: Prespecification in Data Refinement, Information Processing Letters 25, 1987b, pp. 71-76.

Hoare, C.A.R., He, J.: The Weakest Prespecification, Information Processing Letters 24, 1987c, pp. 127-132.

Jones, C.B.: Software Development: A Rigorous Approach, Prentice-Hall International, Inc., London, 1980.

Jones, C.F.: VDM Proof Obligations and their Justification, LNCS 252, Springer Verlag, 1987, pp. 160-186.

Josephs, M.B.: The Data Refinement Calculator for Z Specifications, Information Processing Letters 27, 1988, pp. 29-33.

Kamin, S.: Final Data Types and Their Specification, ACM Transactions on Programming Languages and Systems, vol 5(1), January 1983, pp.97-123.

Morgan, C., Robinson, K.: Specification Statements and Refinement, IBM J. Res. Development, 31(5), September 1987a, pp. 546-554.

Morgan, C.: Data Refinement by Miracles, Information Processing Letters 26, 1987b, pp. 243-246.

Morris, J.M.: Laws of Data Refinement, Acta Informatica 26, 1989, pp. 287-308.

Morris, J.M.: A Theoretical Basis for Stepwise Refinement and the Programming Calculus, Science of Computer Programming 9, 1987a, pp. 287-306.

Morris, J.M.: Specifications for Programming, from Year of Programming Institute on Formal Development of Programs and Proofs, UT Austin, 1987b.

Nakajima, R., and Yuasa, T.: The IOTA Programming System, LNCS 160, Springer Verlag, 1983.

Neilson, D.: Hierarchical Refinement of a Z Specification, LNCS 287, Springer Verlag, 1987, pp. 376-399.

Nipkow, T.: Non-deterministic Data Types: Models and Implementations, Acta Informatica 22, 1986, pp. 629-661.

Polak, W.: Compiler Specification and Verification, LNCS 124, Springer Verlag, 1981.

Spivey, J. M.: The Z Notation: A Reference Manual, Prentice Hall International, 1987.