Part I

Theory
1

Introduction to Data Refinement

1.1 Goal and Motivation

During the process of stepwise, hierarchical program development, a step represents a transformation of a so-called abstract higher level result into a more concrete lower level one. In general, this development process corresponds to increasing the amount of detail required for the eventual implementation of the original specification on a given machine.

In the first part of this book we develop the relational theory of simulation and a general version of Hoare logic, show how data refinement can be expressed within this logic, extend these results to total correctness, and show how all this theory can be uniformly expressed inside the refinement calculus of Ralph Back, Paul Gardner, Carroll Morgan and Joakim von Wright. We develop this theory as a reference point for comparing various existing data refinement methods in the second part, some of which are syntax-based methods. This is one of the main reasons why we are forced to clearly separate syntax from semantics.

The second part of this monograph focuses on the introduction of, and comparison between, various methods for proving correctness of such transformation steps. Although these methods are illustrated mainly by applying them to correctness proofs of implementations of data types, the techniques developed apply equally well to proving correctness of such steps in general, because all these methods are only variations on one central theme: that of proof by simulation, of which we analyze at least 13 different formulations.

We study in particular the similarities and the differences between such widely known methods as the methods advocated by John Reynolds [Ger78, Rey81] and Rick Hehner [Heh93], VDM [J90], Z [Spi92b], and methods more directly based on Hoare logic [Hoa72] or predicate transformers [B78, B88b, Mor89a, Mor90, GM93, MV94, Gar95]. Moreover, we consider methods that,
1.1 Goal and Motivation

although primarily developed for proving correctness of parallel programs, have as their main ingredient a nucleus which essentially concerns sequential (albeit nondeterministic) program refinement, such as the method of Martín Abadi and Leslie Lamport [AL91], and the possibilities mappings of Nancy Lynch [LT87, Lyn90].

This study makes sense because in the final analysis a surprisingly uniform picture emerges when comparing these proof methods — notwithstanding their wide differences in formulation. Either they are special cases of the method based on L-simulation (also called forward or downward simulation), and therefore incomplete as shown in Section 2.2.2, or they are equivalent to the combination of L- and L−1-simulation (also called backward or upward simulation), and then complete by a theorem of He, Hoare, and Sanders [HHS87]. This strengthens our conviction that it makes more sense to emphasize the general principles behind these methods, since we believe these will preserve their value when the students of today have become the specialists of tomorrow, even if present day methods are superseded by more modern ones.

This first chapter presents intuitive explanations and some definitions of key notions in the field of data refinement, such as (abstract/concrete) data type, observability, operation, (data) refinement. The second chapter then introduces the notions of abstraction relation, representation invariant, and simulation, which are used throughout this monograph. In this way the reader is guided to the main questions answered in this part, viz.:

• What is refinement? What is data refinement? What is a correct refinement step, and how can such correctness be proven?

• What is simulation? When one faces a correct case of data refinement, can one always prove its correctness given a particular simulation method? That is, which methods for proving data refinement are complete? Are these methods always sound?

• What is the weakest (i.e., most general) concrete specification simulating a given abstract specification with respect to a given abstraction relation?

• How can one guarantee that termination is preserved during simulation?

The first two chapters serve as motivation and set the scene for the remainder of this monograph, starting with our first technical chapter, Chapter 3. We try to lead gently into the topic and therefore sometimes sacrifice rigor and mathematical precision for intuition.
1 Introduction to Data Refinement

1.2 Introduction to Data Refinement

1.2.1 Basic Issues

Designing a large and complex program usually involves application of some refinement method providing a way to gradually transform an abstract program, possibly a specification, into a concrete implementation. The main principle of such a method is that if the initial abstract program is correct and the transformation steps preserve correctness, then the resulting implementation will be correct by construction. Because an abstract program is, in general, easier to prove correct than a concrete one, this simplifies the structuring of the verification process.

This monograph focuses on methods for proving the correctness of such transformation steps. So we ask ourselves what it means to say that the result of applying a transformation step is regarded as a correct implementation of the construct to which that step is applied. In the context of data refinement, this amounts to the question of when an abstract program $P(A)$ using a data type $A$ is implemented correctly by the more concrete program $P(C)$ obtained from $P(A)$ by replacing operations $A_j$ from $A$ by corresponding operations $C_j$ which belong to a more concrete data type $C$. This question becomes more interesting if we abstract from the particular pair of programs $P(A)$ and $P(C)$ to which this data type transformation is applied. This narrows our subject down to that of data refinement, i.e., formulating when the family of operations $(C_j)_{j \in J}$ belonging to a more concrete data type $C$ correctly implements the family of operations $(A_j)_{j \in J}$ of the more abstract data type $A$. The solution of this problem depends on realizing that $A_j$ and $C_j$ in general constitute programs themselves, which are used as modules inside other, for the moment arbitrary, programs $P$. Now, intuitively, a concrete program module is a correct implementation of an abstract program module, if no program using the concrete module can observe that it is not using the first. That is, “implementation correctness” means that using the concrete program module does not lead to an observation which is not also an observation of the corresponding abstract program module. Note that this does not imply that the concrete and the abstract program display the same observations. There may be observations of the abstract program which are not observations of the corresponding concrete program. Hence, this definition only implies that the observations of the concrete program are contained in those of the corresponding abstract program. This is called refinement.

Example 1.1 (Data refinement step) Consider, for instance, the following two sketches of program fragments in a pseudo-Pascal notation, where $S_1$ and $S_2$
1.2 Introduction to Data Refinement

are dummies for program fragments not involving program variables $U$, $l$, and $x$, and $(\cdot)$ denotes the empty sequence.

\[
\begin{align*}
\textbf{begin} & \quad \textbf{begin} \\
\text{var} & \quad \text{var} \\
U & : \text{set of } N; U := 0; \\
S_1; & \\
U & := U \cup \{x\}; \\
S_2; & \\
y & := \text{a member of } U \\
\textbf{end} & \quad \textbf{end} \\
\text{var} & \quad \text{var} \\
l & : \text{sequence of } N; l := (\cdot); \\
S_1; & \\
l & := \text{append}(x,l); \\
S_2; & \\
y & := \text{first}(l) \\
\textbf{end} & \quad \textbf{end}
\end{align*}
\]

At this point nothing more than an intuitive understanding of the operational meaning of these program sketches is required. Any observable behavior in terms of values of the common program variables $x$ and $y$ exposed by the RHS\(^1\) program should also be a possible behavior of the LHS program. This refinement step comprises of replacing the variable $U$ (ranging over finite subsets of the natural numbers) and operations on it by the sequence-valued variable $l$ and corresponding operations.

Which particular set of observations, i.e., semantics, should characterize a program depends on the particular notion of correctness which a transformation step is intended to preserve. For instance, in the context of relational semantics of programs, this meaning is given by pairs of initial and corresponding final states of its computations. In the case of partial correctness, only terminating computations are represented by such pairs, and then refinement is expressed by inclusion between the corresponding relations. In the case of total correctness nonterminating computations are also made observable through pairs. As explained in Chapter 8, there are then various possibilities for characterizing the meaning of a program relationally, and, consequently, different ways to express refinement.

The above account, suggestive as it may be, introduces a number of terms, those in italics, which need further explanation.

What is an abstract or a concrete program? The notion of abstractness used here is relative. Program refinement distinguishes an upper level, called abstract, from a lower level, called concrete, in order to indicate the direction in which the process of refinement is taking place.

Then, what does “gradually transform” mean in this context? Calling the level at which the transformation starts level 0, this level is transformed into level 1, which may be subsequently transformed into level 2, and so on. Thus,

\(^1\) We use RHS and LHS as acronyms for Right (respectively, Left) Hand Side.
1 Introduction to Data Refinement

A series of successive transformations or *transformation steps* takes effect, until a sufficiently concrete level of implementation has been reached (by some external criterion). As already indicated, we consider this process of transformation to consist of several individual transformation steps, each of which is considered to transform an abstract level (say, level $n$) into a more concrete one (level $n+1$).

In what sense should the terms *data type* and *abstract data type* be understood? Abstract data types are usually defined by a set of operators and a set of axioms, typically given in the form of equations; see e.g. [GH78, LG86, Cle86, BHK89, Par90, Wir90]. As an example of this style we present the equations for an abstract data type $\text{stack}(Z)$, where $\text{Bool}$ refers to the abstract data type of Boolean values (with the usual propositional operators such as $\neg$, $\Rightarrow$, $\lor$, $\land$) and $Z$ to the abstract data type of the elements to be stacked (regarded as primitive at this level of specification).

**Example 1.2 (Characterization of stacks through equations)**

**Name:** $\text{stack}(Z)$

**Operators:**

- $\text{emptystack} : \rightarrow \text{stack}(Z)$
- $\text{push} : Z \times \text{stack}(Z) \rightarrow \text{stack}(Z)$
- $\text{pop} : \text{stack}(Z) \rightarrow \text{stack}(Z)$
- $\text{top} : \text{stack}(Z) \rightarrow Z$
- $\text{empty?} : \text{stack}(Z) \rightarrow \text{Bool}$

**Axioms:**

1. $\text{pop}(\text{push}(z,s)) = s$
2. $\text{top}(\text{push}(z,s)) = z$
3. $\text{empty?}(\text{emptystack}) = \text{true}$
4. $\text{empty?}(\text{push}(z,s)) = \text{false}$

This algebraic characterization of data types is attractive in that it is both program and implementation independent. It merely describes which properties one expects from a data type, not how it is implemented.

In contrast, the, as we call it, model-oriented notion of data types turns out to be less elegant, more implementation dependent, and certainly less abstract.

However, there exist some arguments in favor of the model-oriented approach, which we consider convincing.

First of all there is the essential point whether a method scales up to industrial applications. The methods whose study gave rise to this book, e.g., Reynolds’ method, VDM, and the method by Abadi and Lamport, scale up
1.2 Introduction to Data Refinement

to such applications. These methods can be adapted to concurrency[AL91, MP95, He 89], which is another important point.

Another strong point of model-oriented methods is the characterization of pointer manipulation; it can be easily embedded in the model-oriented style, see, for instance, [AdB94, Mor82].

This does not imply that this algebraic style using equations for defining abstract data types should not be used whenever elegantly applicable. Ole-Johan Dahl’s beautiful monograph [Dah92] is full of examples of data types for cases in which we too consider the algebraic style to be superior. He argues convincingly that for programs using simple data structures whose implementation requires no considerations of efficient memory management, and no carefully defined balance between control flow and data encoding techniques, representation by algebraic terms is preferable because of its simplicity and elegance. This elegance is lost whenever any notion of centralized memory enters the picture, as is, e.g., the case with algorithms for efficient list manipulation or graph traversal. Then other techniques are called for, for instance Reynolds’ method as applied in [LdRG79, vDdR86] for proving correctness of efficient list copying techniques. These techniques do not characterize a data type anymore in the abstract sense using axioms as above. As we shall see, a considerably more application-oriented characterization is used in the context of state-based program verification. This is the reason we reserve the term data type for the latter, and use the terms abstract and concrete data type in the context of refinement steps to indicate the data type on the higher, respectively, lower level of that step [HH90].

Yet even in simple cases the algebraic style requires some elaboration. Assume, for instance, that the stack in Example 1.2 is implemented as an array and a pointer to array cells. This implementation (further investigated in Example 2.6) cannot be proven correct with respect to the above characterization because it does not satisfy axiom 1, as shown in Example 1.3. Although the algebraic method can be extended to solve such problems too, their solution requires an amount of technical machinery which we are not prepared to pay for when facing such a simple, yet fundamental, problem.

But the main reason for adhering to the model-oriented style is that the algebraic approach does not extend very well to concurrency. On the other hand, several of the methods compared here have been exclusively developed with concurrent or distributed systems in mind. These are the methods of Abadi and Lamport and of Lynch presented in Chapter 14. After all, the verification problem is urgent for all but the most trivial concurrent and distributed algorithms. This view is shared by Nancy Lynch in her monograph on distributed algorithms [Lyn96].
1 Introduction to Data Refinement

Example 1.3 Consider an array \( a[1..max] \) and an integer variable \( p \). Let stack \( s \) be represented by the pair \((a,p)\) such that \( 0 \leq p \leq max \), where the contents of \( s \) are stored in \( a[1..p] \), and \( p \) points to the top of the stack. Then \( \text{push}(z,s) \) can essentially be represented by \( p := p + 1; a[p] := z \), and \( \text{pop}(s) \) by \( p := p - 1 \), disregarding for the moment the size of the stack. Consider the case that \( p = 2 \) and \( a[p+1] = 66 \). Execution of \( \text{pop}(\text{push}(77,s)) \) then results in a representation \((a',p)\) of \( s \) different from the representation \((a,p)\) before, since \( a'[3] = 77 \), whereas \( a[3] = 66 \) (see Figure 1.1).

Fig. 1.1. Two different stack representations which cannot be distinguished by using the stack operations.

Hence, this representation of stacks does not satisfy axiom 1. Yet this representation is a common one inside a computer. Verifying correctness of this implementation apparently requires taking more aspects into account than the rather idealized ones listed in Example 1.2.

Notice that the difference between \((a,p)\) and \((a',p)\) cannot be determined by using any of our stack operations (i.e., \text{emptystack}, \text{pop}, \text{push}, \text{top}, \text{and empty?}). Observe that, therefore, such distinctions cannot be expressed in this simple axiomatic style (by its very definition).

Implementations like the one in the example above are called biased. The characteristic property of biased data types is that different representations exist which cannot be distinguished using the data type operations.

We can summarize the above discussion regarding styles by the remark that, in the context of program verification, implementation bias occurs far too often to be ignored. Indeed, our theory deals seamlessly with this concept.

Instead, we shall define data types by means of their models, and so obtain a hierarchy of models when implementing these data types; as stated above, the model at level \( n \) is considered to be more abstract than the one at level \( n + 1 \), which is considered to be more concrete.
1.2 Introduction to Data Refinement

From now onwards we shall resort to an imperative style using state transformations involving the updating of program variables. All the methods studied in this book adhere to the imperative style. This style is called model-oriented. The main difference between this style and the axiomatic approaches is the way operations of data types are specified.

As an example of this style we consider a variant of Example 1.1 in more detail.

Example 1.4 (Representation of sets by lists) Consider the data type of finite sets over some nonempty base type $Z$. We introduce the following operations on a finite-set-valued variable $U$:

- $U := \emptyset$, i.e., assign the empty set to $U$,
- $U := U \cup \{x\}$, provided $x \notin U$ and $x \in Z$, i.e., enlarge the set by the element $x$ from $Z$,
- $x := \text{pick}(U)$, provided $U \neq \emptyset$, i.e., choose a random element from the nonvoid set $U$, delete it from $U$, and assign it to $x$,
- $x \in U$, test whether $x$ belongs to $U$,
- $U = \emptyset$, test whether $U$ denotes the empty set.

Similarly, we introduce the data type $\mathcal{L}$ of linear lists over base type $Z$. We identify linear lists and finite sequences. In general, the set of linear lists $\mathcal{L}$ over a given nonempty set $Z$ is defined as the smallest set $\mathcal{L}$ such that

- the empty linear list $() \in \mathcal{L}$,
- whenever $x \in Z$ and $l \in \mathcal{L}$ then $\text{append}(x,l) \in \mathcal{L}$.

We abbreviate $\text{append}(z_1,\text{append}(z_2,\ldots,\text{append}(z_k,())))$ to $(z_1,\ldots,z_k)$. The following additional operations upon linear lists are needed:

$$\text{first}(\text{append}(z,l)) \triangleq z \quad \text{and} \quad \text{rest}(\text{append}(z,l)) \triangleq l,$$

where $\text{first}(())$ and $\text{rest}(())$ are undefined. Furthermore we need the function $\text{elts}$ recursively defined by $\text{elts}(()) \triangleq \emptyset$ and $\text{elts}(\text{append}(z,l)) \triangleq \{z\} \cup \text{elts}(l)$.

Having defined the set $\mathcal{L}$ of linear lists over a nonempty base set $Z$ of atoms, we proceed with representing the above operations upon a set-valued variable $U$ by operations upon an $\mathcal{L}$-valued variable $l$. Linear list $l$ corresponds to set $U$ if $U$ equals the set of elements contained in $l$, i.e., $U = \text{elts}(l)$. Notice that, in general, this representation is not unique as an element can occur in a linear list more than once, and elements may occur in different order. However, in this particular example insertion of an element $x$ into $U$ only occurs when
1 Introduction to Data Refinement

\( x \notin U \). Since set \( U \) is represented by list \( l \) such that \( U = \text{elts}(l) \), this implies that adding \( x \) to \( l \) only occurs when \( x \notin \text{elts}(l) \). Now we also want to represent the operation of deleting an element from a set. When deleting an element from the corresponding linear list, in general all the occurrences of this element within that list must also be deleted. This is an expensive operation because it requires scanning the whole list. Since efficiency of the implemented operations is an important factor in choosing an implementation, we restrict ourselves to representing sets by linear lists without such repetitions.

<table>
<thead>
<tr>
<th>operation on ( U )</th>
<th>operation on ( l )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U := \emptyset )</td>
<td>( l := \langle \rangle )</td>
</tr>
<tr>
<td>( U := U \cup { x } ) for ( x \in Z, x \notin U )</td>
<td>( l := \text{append}(x, l) ) for ( x \in Z, x \notin \text{elts}(l) )</td>
</tr>
<tr>
<td>( x := \text{pick}(U) ) for ( U \neq \emptyset )</td>
<td>( x := \text{first}(l) ); ( l := \text{rest}(l) ) for ( l \neq \langle \rangle )</td>
</tr>
<tr>
<td>( x \in U )</td>
<td>( x \in \text{elts}(l) )</td>
</tr>
<tr>
<td>( U = \emptyset )</td>
<td>( l = \langle \rangle )</td>
</tr>
</tbody>
</table>

Notice that for a given list \( l \) representing the set \( U \), the nondeterministic operation \( \text{pick}(U) \) is implemented by a deterministic operation, which therefore restricts the nondeterminism inherent in \( \text{pick}(U) \), and is therefore a typical example of refinement.

\( \heartsuit \)

Notice that the example above introduces the notion of representing a data type by another one; we shall soon return to this important concept.

1.2.2 Observability and Refinement

What do the notions observable, observable values, observer mean? The criterion introduced above for the notion of a correct implementation is based on the initial/final state behavior of programs. In our sequential framework the behavior of a program (here, we also consider operations as above to be programs) is supposed to be completely characterized by this behavior, i.e., its relational semantics. Such semantics are defined in Chapter 5 for characterizing terminating computations, and in Chapter 8 in which nonterminating computations are also characterized. Hence one may test a sequential program by providing it with an initial state, and then observing whether a final state is produced, and if so, which one. In such a set-up the program is considered as a black box whose structure (or program text) is hidden from the observer, who has access only to the (initial) values consumed and (final) values produced by that black box, if any. These (pairs of) values are called its observables. As observers programs will again be used. As already remarked above, the