INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Functions, Frames, and Interactions –

completing a λ-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments

Claus Reinke

Bericht Nr. 9804

May 1998

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Functions, Frames, and Interactions – completing a λ-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments

Claus Reinke

Bericht Nr. 9804
May 1998

e-mail: czr@cs.nott.ac.uk
(cr@informatik.uni-kiel.de)

Dieser Bericht enthält die Dissertation des Verfassers

1. Gutachter: Prof. Dr. W. Kluge
2. Gutachter: Prof. Dr. W. Dosch
Datum der Einreichung: August 1997
Datum der Disputation: November 1997

Abstract

The original aim of the work that led to this dissertation was to extend an existing, purely functional language with facilities for input/output and modular programming. The language is based on an untyped λ-calculus, i.e., program execution is defined as program transformation according to a fixed set of reduction rules including β-reduction. Consistently, the implementation comprises an interactive reduction system which is integrated with a syntax-oriented editor: any sub-expression or program result can be submitted for (stepwise) reduction. There is no distinguished main program, no 'global' environment and no explicit static part of the language – in particular, there is no static type system. It is therefore not clear how to add one of the known solutions for input/output or modular programming to such a programming environment. Furthermore, simply adding features to the language would lead to a complex language design with weakly integrated parts, thus losing much of the appeal of purely functional languages.

Help with the latter problem comes from the history of general programming language design: when formal language description techniques were developed and applied to early high-level programming languages, various inconsistencies in the designs of those languages were discovered. To avoid such defects, language design methods based on semantic principles were proposed, such as the principles of abstraction, correspondence and data type completeness. These semantic principles are not biased towards technical details, but rather guide the way from the basic constructs of a language towards a simple and elegant overall language design.

To isolate the fundamental language constructs needed for our particular design problem, we review the support for input/output and modular programming in current functional languages. Surprisingly, we find that most of these languages fall short of adhering to the principles of language design in several respects. We identify some of the problems that result from this fact and argue that, by consistently following the design principles both for the design of the extensions and for the integration of the extensions into the complete language, the weaknesses found to exist in other languages can be avoided. To support this claim, we present a simple language design: we start with a purely functional core based on the λ-calculus, extend it with input/output-facilities and record-like data structures called frames, and complete the language with respect to the design principles.

We go on to show how the resulting design supports a wide range of modular programming techniques and identify the various special purpose constructs used in other languages as instances of a general scheme of abstraction (high-level programming languages also follow this scheme and provide advantages similar

to libraries of pre-defined program components). We conclude that modules, objects and other language constructs for modular programming need only be provided as built-in features in languages which are restricted in their support for general abstraction. This leads to a slightly different view of our proposed language design: λ-calculus should *not* be seen as a part of the functional core language, but rather as providing the means for abstraction over *all* available language primitives which in our case are not only functions, but also frames and interactions.

Our language design features functions, interactions, and modules as first-class data objects, and the input/output-facilities are not restricted to strings of characters, but are applicable to any valid language expression. The latter feature opens a connection between the research areas of purely functional languages and persistent systems, and we argue that both research communities could profit from closer cooperations, avoiding a lot of duplicated work where interests are shared and stimulating and complementing each other where interests differ.

Contents

Chapter 1

Introduction

This thesis focuses on support for input/output and modular programming in purely functional languages. The original problem specification was to extend the reduction language KiR (Kiel Reduction Language [Klu94]) with facilities for input/output and modular programming. The language is derived from an untyped λ-calculus, and program execution is defined as program transformation according to the reduction rules of an extended λ-calculus. In this thesis, we present our work as a language (re-)design process, guided by principles that are explained below. We briefly summarize the design decisions that led us to consider functional languages, and review the existing work and design options in the fields of input/output and modular programming in these languages. On this basis, we develop a simple language design that solves the original problem, extending a functional core language with facilities for input/output and modular programming. As far as possible, we abstract from the peculiarities of KiR and its implementations [GK96], but the λ-calculus part of the language turns out to be essential for our design.

Purely functional languages, and reduction languages in particular, represent a radical departure from the von Neumann model of programming (cf. the discussion in [Bac78]). Initially, programming centered upon the problem of controlling the dynamic behavior of some computer equipment. In order to utilize early programmable computers for the solution of abstract problems, it was not sufficient to devise an abstract algorithm. Programmers had to map the abstract algorithm to a machine program and the initial problem conditions to an initial machine state. After a computation, they had to retranslate the final machine state and the intermediate outputs into their abstract problem domain. Later, large parts of the two-way mapping between machine states and objects in the problem domain were delegated to the computer, too. The resulting imperative programming languages allowed to specify computations on a higher level of abstraction which was hopefully closer to the problem domain and certainly farther away from the details of the machines. While these languages provide a more abstract view of machine states, imperative programming still involves a

user-supplied mapping of abstract algorithms into explicitly specified sequences of (abstract) state transformations.

In contrast, the very essence of *declarative programming* is usually described as specifying *what* is to be computed instead of *how* this should be done, eventually leaving even the mapping of a problem specification to a computation to the computer. Various forms of declarative languages have been developed, but we only deal with *functional languages* here or, more precisely, with *reduction languages* (cf. [KS86, Klu92]). The idea is to start with some calculus and its rewrite (reduction) rules, and to extend it to a general purpose programming language whose semantics is directly and completely defined by the reduction rules of the extended calculus. For instance, the λ-calculus [Chu51] does already correspond to a basic functional programming language and can be extended consistently to practical ones (cf. [Lan63, Ber76, BF82], among others). The fundamental β-conversion rule of the λ-calculus can be employed, in its directed form of β-reduction, to define a *reduction semantics* for the extended functional languages. The execution model that results from such a reduction semantics is based on *high-level meaning-preserving program transformations*.

Based on program transformations instead of state transformations, these languages support a declarative style of programming (programs that correspond to problem descriptions are transformed to programs that correspond to problem solutions), and the reduction semantics lends itself to implementations that map program transformations to transformations of machine states. Nevertheless, programs are running on state transforming machines embedded in a real world, and it is in this setting that modern functional programming languages, due to their high level of abstraction, encounter some compatibility problems with these machines. One of the problems is fairly obvious: the higher the level of abstraction, the more difficult the mapping of programs to efficiently executable machine programs becomes. Efficient implementations will continue to be a major research topic though considerable progress has already been made in this respect, as a consequence of which functional languages are more and more being used for practical applications. This, in turn, brings up some pragmatic considerations that are the main topics of this thesis.

We focus on two seemingly unrelated problems, solutions to which are essential if functional languages are to be used in everyday programming practice. The first one is an immediate consequence of the high level of abstraction of functional languages: how must interactions between a program and an external environment (consisting of, e.g., input/output-devices, file systems, ...) be described in a programming language that abstracts from the existence of an outside world? This problem affects communication with users as well as explicit control over the state of the computing equipment and the programming environment or any other state- or communication-based computation. Without an adequate solution, functional languages could hardly be called general purpose. In imperative languages, explicit modification of a global system state is the only way of doing

anything, so they seem to be well suited for this kind of applications. Functional languages, on the other hand, abstract from the underlying machinery, which seems to be counterproductive in this case, but conforms to the idea of declarative programming and has proven to be very useful in general.

A more accurate description of the problem is that application domains are not always far away from the peculiarities of real machines, and sometimes the *how* of computation is exactly *what* needs to be specified. Of course, this does not imply that declarative programming is the wrong approach, and we certainly do not want to go back and make every program depend on the details of the machines it is supposed to be running on. Instead, it is necessary to develop declarative means to describe (some of) the details of machine states or of state changes and to find ways to let abstract programs interact with real machines and runtime environments. These interactions should only be used when the problem domain demands it, but should not permeate the complete programming language, as an imperative add-on for a declarative language would certainly do. The goal is to find a declarative way to describe inherently state- or communication-based computations in a certain class of problem domains.

The second problem, one that functional programming languages share with every other general purpose language, is that programs may grow and that managing the complexity of large programs requires appropriate language support. The very first concept needed to cope with large programs is abstraction. Organizing a system of objects into layers of abstraction, concentrating only on the objects relevant to a particular layer at a time, keeps the complexity of each layer manageable. In contrast to conventional programming languages, functional languages do already provide an additional layer of abstraction, as they completely liberate programming from organizing program execution on the underlying machinery. It is therefore reasonable to assume that the functional paradigm can cope with problems at least one order of magnitude more complex before the program sizes become unmanageable. Moreover, it is the very idea of functional programming to make extensive use of (functional) abstraction and to compose complex programs systematically from smaller ones, which should render the transition from programming-in-the-small to programming-in-the-large much less problematic than in conventional languages. Indeed, there seems to be no reason why the complexity of programs should have any influence on the functional style of programming, or why certain language constructs should be useful only for large programs. So, if functional languages do really encounter any problems related to complexity, but not to problem size, there may be some fundamental features missing in functional languages, and large programs are only one problem domain where these shortcomings unavoidably manifest themselves.

We argue that this is indeed the case: functional languages treat programs as data objects and are essentially well suited for modular programming, but they do not provide sufficient support for the manipulation and organization of large collections of long-living data objects. Large collections of program mod-

ules exhibit exactly these characteristics, which is the real reason why functional languages cannot bring their intrinsic qualities into play for modular programming. Thus, the concept of modular specifications needs to be (re-)examined in the context of functional programming languages. At the very least, it should be possible to organize large programs into smaller, more comprehensible modules that can be stored, modified and reused independently of each other.

If we want to overcome these deficiencies, we have to face the problems of either designing a new functional language or of making major extensions or modifications to an existing one. In either case, an essential part of our work is related to language design [Lea93, Was80]. In this thesis, we intend to build on an existing reduction system and may thus focus on the extensions that are necessary to support interactions with runtime environments and programming-in-the-large, but it is still advisable to keep the complete language in mind in order to avoid inconsistencies in the overall design.

Fortunately, several principles have been established to guide the design of new languages as well as the evaluation and the redesign of existing ones. We use here three semantic principles ('principles derived from the denotational approach to programming language semantics' [Ten77]), collected by Tennent and Morrison [Ten77, Mor79]. Morrison points out that *'the overall design aim of power through simplicity, simplicity through generality should be the guiding light'*[1] and goes on to state the principles of abstraction, correspondence, and data type completeness, the origins of which can be traced back to [Lan66, Str67]. We summarize the descriptions from [Ten77] and [Mor79]:

principle of abstraction *Abstractions should be allowed over all semantically meaningful syntactic categories of a language.*

principle of correspondence *The rules governing names in a language should be designed together in order to avoid irregularities in the manner in which the names may be used. In particular, there should be a one-to-one correspondence between declarative and parametric forms to introduce names.*

principle of data type completeness *All data types should have the same 'civil rights' and the rules for using data types should be complete without exceptions.*

Although not specifically intended to guide the design of functional languages (they have first been used to evaluate the design of Pascal [Ten77] and to design a variant of Algol [Mor79]), the principles are in perfect conformance with the properties of these languages[2]. For instance, the first two principles are concerned with the consistent use of names and the predominant role of abstraction

[1] paraphrasing van Wijngaarden [vW63]

[2] which is no surprise considering the origins of the principles

which is also what the λ-calculus (the basis of many functional languages) is all about. The third principle is a nice generalization of one characteristic feature of functional languages, namely that functions are first-class data objects. While this term may be misleading insofar as it insinuates that functions are treated as something special, the principle of data type completeness emphasizes that it is the elimination of (unnecessary) special rules which is important. It simplifies the language, makes it more complete and thus more expressive. Similarly, the language is less complex if there are no distinctions between names introduced by (local) declarations and names introduced as formal parameters of abstractions.

In addition to these general design principles for programming languages, some additional design constraints derive directly from our decision to use purely functional languages. Briefly, this class of languages has very useful properties (e.g., referential transparency), and we want our extensions to be *conservative* with respect to these properties. Since we start our design process with a core language that complies with the general design principles, both the principles and the particular properties of purely functional languages emphasize characteristics of the language that we should try to conserve. In other words, these properties define the invariants of the language modification process. While these constraints are sometimes quite restrictive, they help to avoid solutions which, on first glance, appear to be simple but may cause serious problems in the long run.

Finally, we can generalize from the history of programming languages [Weg76, Wex81, Wex93] to get an intuition of the situations that occur in language design processes. General purpose programming languages are not static, but are continually extended and developed in response to ever evolving user requirements. In the long term, the language complexity increases with the requirements, but this process is not continuos in general. Whenever a new problem area must be dealt with, the design space is first explored with a large number of language extensions and variations. Experience with these experimental language modifications may lead to a better understanding of the original problem and thus to simpler solutions and a decrease in language complexity. Otherwise, the language is growing with every new extension until it becomes so complicated that someone decides to make it simpler at any price, cutting out the least useful features, developing generalizations for groups of features or developing a special purpose variant. Furthermore, design goals are not always conflict-free, and every set of design decisions corresponds to a choice in favor of some of the goals. If more than one of the possible choices is realized, the original language design splits up into a family of more or less related languages (domain-specific languages are just a special case of this). These different design decisions need not be made at the same time, indeed, language designers may become aware of their choices only after following one particular path for a long time (for instance, imperative languages dominated the stage before it became feasible to implement declarative languages). Ideally, the different variants can be combined again by some generalization step which may be discovered after some time of experimentation

with each of the variants. Several instances of this general picture have been encountered in the design process described in this thesis.

The main goals of this work are to develop an input/output-system and a module system for the reduction language KiR [Klu94], derived from an untyped λ-calculus, which seems to involve two major extensions to the existing reduction system. However, by following the design principles faithfully, it turns out to be possible to achieve these ends by rather minor extensions of the language, keeping most of the added complexity on the implementation level. Moreover, since the simplicity of the extended language stems from generalization, not from restriction, the resulting language, called FFI (for functions, frames, and interactions) is not only simpler, but also more expressive (in the areas of input/output and modular programming) than other current functional languages [Hud89]. We show that conventional approaches to the problem areas of input/output and modules exhibit major inconsistencies with respect to the principles stated above and that the approach taken here avoids these inadequacies.

That a seamless integration of input/output into purely functional languages would be possible at all was only discovered in the late 1970s (with implementation work starting around 1980), and various solutions, based on streams, continuations or, more recently, on monads or uniqueness types, have been proposed and implemented since then (cf. the summary in section 3.6). Most of them, however, require further language support for features that are not essential to input/output, e.g., for non-strict data constructors, lazy evaluation or static type systems. Theoretical frameworks have been developed that allow to compare these alternatives and to prove their equivalence in expressiveness [Gor94, HS89], but the relations between the approaches are usually presented from a historical or from a theoretical perspective, and the situation is still unsatisfactory with respect to language design. The problems of input/output in functional languages are therefore reviewed with a stronger emphasis on their pragmatic aspects here, leading to a more uniform and comprehensive presentation of the subject, which provides the necessary support for the design decisions to be made later.

The other main research area involved, that of module systems for functional languages, is even more divergent than the area of input/output. Beyond the very basic idea of modules as program building blocks, there seems to be no agreement on basic terminology, not to speak of problem specifications or solutions (cf. chapter 4). The approaches include the simple adoption of module systems for conventional languages, the interpretation of modules as types (and types as values), modules as records, modules as (first-class) environments, modules as data abstractions or even the abandonment of modules in favor of object-oriented language extensions. The main pragmatic differences have to do with the expressiveness and completeness of the module languages, and whether the module language and the programming language should be separated or not. An overview over the basic ideas underlying some of the existing or proposed module systems for functional languages is developed, and the pros and cons of the

various design choices are examined.

Equipped with suitable problem specifications and general surveys of the possible design choices in each area, the next step is to actually build the design according to the design goals stated above, keeping an eye on the possible interactions between the three areas of the functional core language, the input/output-system and the module system. The first surprising encounter is that almost all currently used input/output-systems for functional languages violate the principle of data type completeness because the set of objects that may be communicated is usually restricted to strings of characters (cf. also [HMST92]). All other objects, including data structures and functions, are no longer first-class citizens when it comes to input/output. We decide to abandon these restrictions and to allow all objects of the language to be communicated. While this decision simplifies the language, it makes high demands on the implementation. However, it turns out that we do not have to start from scratch here. Morrison et al. [ABC+83a] came to a similar design decision when trying to integrate high-level imperative languages and database systems. They identified *'persistence as an orthogonal property of data, independent of data type and the way in which data is manipulated'* and founded the research area of *orthogonally persistent programming languages and systems*. Interestingly, Morrison did not address this in his thesis [Mor79, chapter 3.7], but noted: 'the file system functions do not act on all data types which breaks the principle of data type completeness. This is a strong indication that more work is required on this problem.'.

So, as an immediate consequence of making similar design decisions, our approach to input/output demands a connection between purely functional languages and the area of persistent programming languages and systems, which is by now well researched. We briefly review the reasons why, until recently, persistent and functional languages have been two separate areas of research and argue that this is a very unfortunate situation. Recent improvements of functional languages and their implementations, especially in the area of input/output, have eliminated the main obstacles to a combination, and both fields could profit from a cooperation. To name one example closely related to this thesis, persistent languages are particularly promising for the construction of integrated programming environments, thus guiding the way towards further developments of better programming environments for functional languages.

With respect to the module system, we argue that, in order to facilitate the reuse of modules in the construction of new programs, the module language should not be unnecessarily restricted. At least at the level of program construction, modules form a semantically meaningful category and abstraction should therefore be allowed over modules. If abstraction and composition are the important features required of a module language, it seems natural to use a functional language for this purpose. We even go one step further and integrate the module language and the functional programming language. Again, this simplifies the language design, but this time, the necessary extensions to the implementation

are also small. Merely for convenience, we add *frames* to the language, represented as record-like data structures associated with a set of primitive operations. Applying the principles of abstraction and data type completeness again, all syntactically legitimate objects may be placed in frame slots and, as abstractions over frames are allowed, frames themselves may be passed as arguments to functions or returned as their results. In combination with the new input/output-system, frames containing functions may be stored in files and retrieved from there to become parts of other programs, giving all the flexibility needed for a versatile module system without the complexities that usually come with it.

We have consciously decided to maintain the implicitly and dynamically typed nature of our reduction language, thus avoiding the restrictions of static type systems in our design. Current static type systems (usually extensions of [Mil78]) are still unsatisfactory due to the constraints they impose on expressiveness, and the various lines of research have not yet culminated in a stable and uniform framework (cf. also the discussion in section 8.2). Indeed, there is evidence that some of the ideas used here were available more than a decade ago, but were not widely used because they did not fit in with the then popular type inference systems. These attempts unveiled a number of inadequacies in early type systems and initiated further research work there. Using an almost untyped framework here avoids the duplication of research efforts and enables a better separation of concerns, as the results presented here do not depend on the availability of any particular type system. Still, it is necessary to mention some of the typing problems and (partial) solutions, both to give a fair and complete description of related work and to point out where the language design might have been restricted by the constraints of a static type system.

The thesis is organized into two parts. The first one explores the foundations of this work, trying to separately identify requirements, design options and design constraints for input/output and module systems in the context of functional languages. In chapter 2, we present functional languages as a variant of declarative programming languages, also giving a brief review of their formal bases. Chapter 3 deals with the problem of letting functional languages interact with their runtime environments, and chapter 4 explores the problem area of module systems for functional languages. Both chapters 3 and 4 include overviews of existing work and references.

The second part builds on the design framework established in part one, and proposes one particular language design which combines ideas from all research areas discussed in the foundation part. The formal language definition is given in chapter 5, chapter 6 investigates the issues of modular programming in this language, and some interesting aspects of an implementation of the design are described in chapter 7. Chapter 8 shows options for further work, relates our research to that on persistent languages and systems, and discusses some of the problems of translating our design into a typed framework.

Chapter 9 contains a summary and conclusions.

Part I

Foundations

Chapter 2

Functional Programming Languages

Programming may be understood as a discipline which describes the static properties and the dynamic behavior of existing or imagined systems in order that a computer may be used to analyze, simulate or control them.

2.1 From rule-based transformation systems to reduction languages

Given a specification of the parts of a system, its dynamic behavior may simply be specified by a set of transformation rules, each of which defines what the objects before and after a computation step are. Depending on the universe of discourse, the objects may be terms, graphs, system states or logical formulas, and the corresponding variants of transformation are known as term or graph rewriting, as state transformations or as deduction systems, respectively. For general surveys of the field (as well as further references and proofs), see [Der93, Klo90, DJ90].

Given a set Obj of objects, a set of transformation rules T could be specified explicitly as a subset of $Obj \times Obj$, writing $A \mapsto_T B$ for $(A, B) \in T$, but this would be tedious. For a more concise specification, it is helpful to factor objects into contexts C_{Obj} and sub-objects, where contexts are objects with holes in which (sub-)objects can be placed to form complete objects again:

$$\forall O \in Obj, C \in C_{Obj} : C[O] \in Obj.$$

It is then possible to abstract over contexts or sub-objects in rules, writing

$$\forall C \in C_{Obj}, X \in Obj : C[C_A[X]] \mapsto_T C[C_B[X]],$$

for given contexts C_A and C_B, as a finite representation of a possibly infinite set of rules, where neither the sub-objects (X) nor the embedding contexts (C) are

modified. Since rules that abstract over constant contexts occur frequently, the abbreviation $\rightarrow_T$ is used for these *context-free* (or context-independent) substitutions:

$$\forall A, B \in Obj : (A \rightarrow_T B =_{def} \forall C \in C_{Obj} : C[A] \mapsto_T C[B]).$$

Furthermore, qualification of objects and contexts may be used for finer control over the specified set of rules.

The execution model of object transformations is quite simple: for a given object A, choose a rule $A \mapsto_T B$ and replace A with B (this may involve instantiations of contexts and sub-objects in rules that abstract over these). If there is no such rule, A is *irreducible* and represents a *result* of the computation

$$\forall A, R \in Obj : (A \downarrow_T R \Leftrightarrow_{def} ((A \mapsto_T^* R) \wedge (\neg \exists R' \in Obj : R \mapsto_T R'))).$$

If $A \mapsto_T^* R$, R is said to be *derivable* from A using T. However, this purely operational view of programming is hardly sufficient, for if the only way to understand a program is to execute it (mentally or otherwise), writing programs is a rather clueless activity. The rule abstractions used above do help here, as they allow programs to be generalized, so that the same transformation system may be used for different sub-objects and in different contexts. The notions of reducible and irreducible objects and the use of the transitive reflexive closure of T are attempts to understand transformation systems in terms of the possible results they can generate for given objects, abstracting from the computation steps in between. It is possible to classify transformation systems according to their possible results. A system is *terminating* if every computation terminates (every possible transformation sequence is finite). A system is *confluent*, if any two computations starting from a given object can be extended to reach a common object:

$$\forall A, B_1, B_2 \in Obj : (A \mapsto_T^* B_1 \wedge A \mapsto_T^* B_2) \Rightarrow \exists C \in Obj : B_1 \mapsto_T^* C \wedge B_2 \mapsto_T^* C.$$

In a confluent system, every terminating computation on a given object yields a unique result (the set $\{\{R | A \downarrow_T R\} | A \in Obj\}$ contains only singletons or empty sets). Note that, even in a confluent system, for a given object there may be many different sequences terminating with the same irreducible object and other sequences not terminating at all.

When reasoning about systems that go through sequences of changes, it is often helpful to consider invariants, i.e., properties of the system that do not change. If sensible invariants can be established for a given set of transformation rules, it follows that an equivalence relation can be defined on the set of objects in such a way that transformation steps never change the equivalence class of objects (with respect to the invariants, they only *replace equals by equals*). In the

simplest case, the equivalence relation is completely defined by the transformation rules:

$$\forall A, B \in Obj : A =_T B \Leftrightarrow_{def} (A, B) \in (T \cup T^{-1})^*.$$

If a transformation system supports a sensible equivalence relation, it can be used to provide a declarative view of the system. The transformation of any given object will either terminate with an equivalent and irreducible object or not at all, and thus the only purpose of any computation on an object is to find irreducible representatives of the equivalence class inhabited by the original object. If all members of the same equivalence class are considered indistinguishable (*semantically equivalent*), the differences between the intermediate objects of a computation, and thus the computation itself, are simply not observable (with the important exception of non-termination), leading to a complete abstraction from the operational view. Considering all intermediate objects as distinct leads to a fully operational view as explained above. The possibility to devise mixtures between these two extreme views and thus to focus on as much operational detail as necessary for a specific purpose seems to be very attractive. In particular, it renders the idea of declarative systems practical: the operational view, though not inherently tied to specific machinery, provides enough detail to enable implementations of a given transformation system, but is not very suited for reasoning about the system. Declarative views of the same transformation system arise merely as abstractions from operational details, are more suited for reasoning but less so as the basis of an implementation. Both views are necessary and can be based on the same specification.

Let Obj_0 be the set of strings built from $\{true, false, obj, \vee\}$.

$$\begin{aligned} obj &\rightarrow_0 true \\ obj &\rightarrow_0 false \\ obj &\rightarrow_0 obj \vee obj \end{aligned}$$

Let $Obj_1 =_{def} \{X | obj \downarrow_0 X\}$. $\forall X \in Obj_1$:

$$\begin{aligned} true \vee X &\rightarrow_1 true \\ false \vee X &\rightarrow_1 X \end{aligned}$$

Figure 2.1: A simple two-level transformation system

This approach does not immediately exclude non-confluent transformation systems, although, for transformation systems with non-deterministic results, viewing the set of rules as a description of an equivalence relation often runs counter to 'natural' interpretations. As an example, consider the simple system

of figure 2.1, which describes a logic of disjunctive terms with a sequential disjunction. The set of terms is itself specified as a transformation system, in the style of a Backus-Naur form, with the terminal strings being just the irreducible objects derivable from the start symbol, *obj*. In this particular case, any equivalence relation based on the rules of $\rightarrow_0$ has $false$ and $true$ in the same equivalence class, but only in that both are derivable from the same non-terminal object, not in any 'logical' sense. Moreover, the system could be reformulated using an equivalence on sets instead of a partial order ($\rightarrow_0$) on elements, collecting the alternatives for the left hand side *obj* on the right hand side of one rule instead of giving many rules with the same left hand side

$$Obj = \{true\} \cup \{false\} \cup \{X \vee Y | X, Y \in Obj\}.$$

A similar trick can be employed for all non-deterministic systems by working with the powerset $\mathcal{P}(Obj)$ of objects instead of Obj itself.

Static descriptions of objects will still be given in the style of an extended Backus-Naur form, but the description of dynamic transformations is simplified substantially if only confluent systems are considered. In these, there is no need to search for all irreducible representatives of an equivalence class, since any class which has an irreducible representative is uniquely determined by this one object. Furthermore, there is no inherent exponential overhead due to the use of sets of objects, and equivalent objects can be transformed into the same representative (*Church-Rosser-property*):

$$\forall A, B \in Obj : A =_T B \Rightarrow \exists C \in Obj : A \mapsto_T^* C \wedge B \mapsto_T^* C,$$

so there is no need to apply transformation rules backward and forward in order to prove two objects equivalent.

Providing the full range of possibilities offered by object transformation systems for the specification of programs is, in general, not advisable. One problem with user-specified transformation systems is that they may be non-confluent and that it is not even decidable whether or not they are. Furthermore, the equivalence generated by the transformation rules may be inconsistent (requiring all objects to be in one class), thus offering no help at all for reasoning about programs. Fortunately, restricted classes of transformations can be given that turn out to be expressive enough for almost all conventional programming tasks and guarantee that some useful properties cannot be invalidated unintentionally by programmers.

A simple way to get a functional language with user-specified rules is by restricting the set of possible rules, e.g., by requiring that each left hand side has to conform to the pattern `f(`$\mathtt{t}_1$`, ..., `$\mathtt{t}_n$`)`, where `f` is called a function symbol and the $\mathtt{t}_i$ are constructor terms, i.e., variables or terms of the form $\mathtt{c}_i$`(`$\mathtt{t}_{i1}$`, ..., `$\mathtt{t}_{im_i}$`)`, where the $\mathtt{c}_i$ have no defining rules and the $\mathtt{t}_{ij}$ are constructor terms.

Further restrictions include that all variables that occur on the right-hand side of a rule must also occur on the left-hand side of the same rule, and that no variable may occur twice on the left-hand side of the same rule. By construction, no left-hand side of one rule can then overlap (share instances) with a proper (non-variable) subterm of any other rule (except itself). These *orthogonal systems* are known to be confluent irrespective of termination. However, this is a pure strategy of restriction, achieving useful properties only by discarding problematic classes of transformation systems.

It is much more helpful to enrich transformation systems with a (mathematical) theory, e.g., equational logic for transformation systems supporting an equivalence relation or predicate calculus for inference rules. As these calculi can themselves be defined by rewrite rules, another approach to get a functional language is to define confluent rewrite rules for a suitable calculus of functions and take this as the basis for a programming language. Confluence and other useful properties can then be guaranteed by the language designer, who also has to make sure that the calculus, in spite of the fixed set of rules, provides a sufficient foundation for an expressive programming language. This approach has some definite advantages, as it offers the full range of transformation systems to the language designer, who provides a programming language based on a calculus as a high-level interface to the general execution model of object transformations. Programmers can just use these high-level interfaces, which are defined through the general model, but come equipped with an additional theory that may be used to reason about programs. This way, programmers gain something for the flexibility they loose.

Note the subtle difference here between mapping a given language into a calculus and developing a language as an extension of a calculus. The former is the approach of denotational semantics [Sto77], which tries to give mathematical models for existing languages in order to make their informal semantics exact in a more or less common framework and to enable comparisons and evaluations of these languages. The latter describes the approach taken here in this thesis, which starts with a formal system and develops this core into a suitable language without sacrificing any of the characteristic properties. Language design is thus constrained, and it is often difficult to invent extensions that satisfy both these constraints and the pragmatic requirements for a general purpose programming language. However, we have found that the formal constraints establish a valuable counterpart to the evolving but usually poorly specified user requirements. They provide a more balanced environment for language design and help to guide the design process away from shortsighted, problem-specific language extensions and towards general solutions that keep the language simple and general. Language designs that are developed without this kind of guidance often get more and more complex with every new feature they include to please their user community. Finally, the interactions between the short-term solutions lead to problems in the long run.

There are many calculi that could be used as a starting point for a functional language, including combinator-calculi and λ-calculi, which have been invented to build foundations of mathematics based on functions [Chu51, CF74, Bar84, HS86], or the more recent and more implementation-oriented supercombinator-calculi [Hug82]. The focus is on λ-calculi here since even the basic λ-calculus provides elements that correspond to function definitions and applications, to abstraction, static name binding and nested scopes. Despite the simplicity of the calculus, this is a rich foundation to start from when designing a functional language. In fact, the semantics of functional languages can be defined directly and completely through the reduction (transformation) rules of such calculi, and the languages that result from such an approach are called *reduction languages*.

2.2 λ-calculi

$\forall x \in var,\ n, k \in Nat$:

$$
\begin{array}{rcl}
term & = & \backslash^n\ x \\
 & | & \lambda x.term \\
 & | & (term\ term)
\end{array}
$$

$$
\begin{array}{rcl}
BC_x^0 & = & [\ ] \\
BC_x^n & = & \lambda y.BC_x^n\ ,\ if\ y \neq x \\
 & | & (BC_x^n\ term) \\
 & | & (term\ BC_x^n) \\
BC_x^{n+1} & = & \lambda x.BC_x^n
\end{array}
$$

$$
\text{the occurrence of x in } BC_x^n[\ \backslash^k\ x\ ] \text{ is } \begin{cases} bound & if\ k < n \\ free & if\ k = n \\ protected & if\ k > n \end{cases}
$$

Figure 2.2: Syntax and binding structure of a basic λ-calculus

The basic λ-calculus presented here is a modification of Church's original λK-calculus [Chu51, chapter V, section 17]. The syntax is given in the upper part of figure 2.2 where *var* is a non-empty set of distinguishable identifiers (*variables*). A λ-*term* is either a variable preceded by a number of *protection keys*, a λ-*abstraction* binding a variable in an *abstraction body*, or an *application* consisting of an *operator*- and an *operand*-term. The essence of the modification,

proposed by Berkling in [Ber76], is the use of protection keys as a means to protect variables against immediately enclosing bindings. Compared with λK, it is slightly more complicated in formal treatment but better suited as the basis of practical programming languages since it keeps both the binding structure and the original variable names of terms intact while performing β-reductions.

The binding structure of the modified calculus can be formalized using binding contexts as defined in the second part of figure 2.2. A binding context BC_x simply counts the number of λx-abstractions in which its subterm is embedded (the context is uniquely determined by the term, the subterm and the variable name). In contrast to λK, a variable occurrence is not necessarily bound to the next enclosing λ-abstraction with the same name. In particular, the occurrence of x in $\lambda x.BC_x^n[\ \backslash^n\ x\ ]$ is bound to the outermost λx (independent of the exact structure of BC_x^n, the n protection keys cancel out the n enclosing bindings in BC_x^n).

The fundamental operation in all λ-calculi is the substitution of a λ-term for free occurrences of a bound variable. In the original calculus, this requires the renaming of bound variables to avoid name-clashes that would occur if free occurrences of a variable are substituted into the binding scope of a λ-abstraction with the same variable name. In the modified calculus, protection keys are used to keep the binding structure intact without changing any variable names. Substitution is defined in figure 2.4 using an auxiliary function Π (cf. figure 2.3) to modify protection keys. Π_x is a simple recursion on the structure of λ-terms, counting the number of λx encountered while descending and acting only on free and protected occurrences of x. $\Pi_{x,0}^{+1}\ term$ is used to protect all free and protected occurrences of x in $term$ against an additional λx-abstraction, whereas $\Pi_{x,1}^{-1}\ term$ is used to remove one level of protection from all protected occurrences of x in $term$ whenever an intermediate λx-abstraction disappears. Similarly, $[\backslash^k x \leftarrow N]M$ replaces all k-fold protected occurrences of x in M by N. Free and protected occurrences of variables in N are protected against inner bindings in M using Π while descending into the body of an abstraction. The protection of the variable to be substituted is also adapted to avoid substitutions of bound variables.

As specified here, both substitution and the auxiliary operation Π are defined through terminating and confluent transformation systems whose irreducible objects are λ-terms but whose initial and intermediate objects are not. Their transformation rules are not part of the λ-calculus, but applications of Πs and of substitutions to λ-terms are used as meta-notation representing their results. Using these auxiliary operations, the transformation (reduction) rules of the λ-calculus are given in figure 2.5. α-conversion allows renamings of bound variables, β-reduction replaces an application with an abstraction in operator position by the abstraction body, where all free occurrences of the bound variable are substituted by the term in operand position. Finally, η-conversion identifies abstraction as a kind of inverse term-forming operation to application, as long as no variables are captured. The complementary equivalence, establishing application as an in-

$\forall x, y \in var,\ M, N, P \in term,\ n, k, j \in Nat,\ m \in Int:$

$$
\begin{array}{lcll}
\Pi^m_{x,n}\ \backslash^k x & =_\Pi & \backslash^{k+m} x & ,\ if\ k \geq n \\
\Pi^m_{x,n}\ \backslash^k x & =_\Pi & \backslash^k x & ,\ if\ k < n \\
\Pi^m_{x,n}\ \backslash^k y & =_\Pi & \backslash^k y & ,\ if\ y \neq x \\
\Pi^m_{x,n}\ (MN) & =_\Pi & (\Pi^m_{x,n}\ M\ \Pi^m_{x,n}\ N) & \\
\Pi^m_{x,n}\ \lambda x.M & =_\Pi & \lambda x.\Pi^m_{x,n+1}\ M & \\
\Pi^m_{x,n}\ \lambda y.M & =_\Pi & \lambda y.\Pi^m_{x,n}\ M,\ if\ y \neq x &
\end{array}
$$

Figure 2.3: Modification of protection keys

$$
\begin{array}{lcll}
[\backslash^k x \leftarrow N]\ \backslash^k x & =_\sigma & N & \\
[\backslash^k x \leftarrow N]\ \backslash^j y & =_\sigma & \backslash^j y & ,\ if\ (j \neq k) \vee (y \neq x) \\
[\backslash^k x \leftarrow N]\ (MP) & =_\sigma & ([\backslash^k x \leftarrow N]M\ [\backslash^k x \leftarrow N]P) & \\
[\backslash^k x \leftarrow N]\ \lambda x.M & =_\sigma & \lambda x.\ [\backslash^{k+1} x \leftarrow \Pi^{+1}_{x,0}\ N]M & \\
[\backslash^k x \leftarrow N]\ \lambda y.M & =_\sigma & \lambda y.\ [\backslash^k x \leftarrow \Pi^{+1}_{y,0}\ N]M\ ,\ if\ y \neq x &
\end{array}
$$

Figure 2.4: Substitution of λ-terms for free variables

$$
\begin{array}{rcl}
\lambda x.M & =_\alpha & \lambda y.\Pi^{-1}_{x,1}\ [x \leftarrow y]\ \Pi^{+1}_{y,0}\ M \\
(\lambda x.M\ N) & =_\beta & \Pi^{-1}_{x,1}\ [x \leftarrow \Pi^{+1}_{x,0}\ N]M \\
M & =_\eta & \lambda x.(\Pi^{+1}_{x,0} M\ x)
\end{array}
$$

Figure 2.5: Conversion rules of the λ-calculus

verse to abstraction if no variables are captured, is an immediate consequence of β-equivalence:

$$
\begin{array}{rcl}
(\lambda x.\Pi^{+1}_{x,0} M\ x) & =_\beta & \Pi^{-1}_{x,1}[x \leftarrow \Pi^{+1}_{x,0} x]\Pi^{+1}_{x,0} M \\
& =_\Pi & \Pi^{-1}_{x,1}[x \leftarrow \backslash x]\Pi^{+1}_{x,0} M \\
& =_\sigma & \Pi^{-1}_{x,1}\Pi^{+1}_{x,0} M \\
& =_\Pi & M
\end{array}
$$

It is important to note that the modifications only affect the representation of λ-terms and rules whereas, as Berkling formulated it in [Ber76], the conceptual substance of the λ-calculus is not changed. The modifications are consistent with the original calculus and do not change any equivalences between its terms (which are also terms of the extended calculus). This postulates a rather strong relationship between λ_{Church} and $\lambda_{Berkling}$ which can be formalized as follows.

Theorem 2.1 (Correspondence between λ_{Church} and $\lambda_{Berkling}$)
Let the subscripts C and B denote terms and rules in λ_{Church} and $\lambda_{Berkling}$, respectively and let $var_C = var_B$ be an infinite set of variable names. Then

1. $term_C \subset term_B \wedge \forall M_C, N_C \in term_C : M_C =_C N_C \Leftrightarrow M_C =_B N_C$

2. $\forall M_B, N_B \in \{M \in term_B \mid$ *no variable occurs protected in* $M\} :$
   $M_B =_B N_B \Rightarrow$
   $\exists M_C, N_C \in term_C : (M_C =_{\alpha_B} M_B) \wedge (N_C =_{\alpha_B} N_B) \wedge M_C =_C N_C$

Proof:

1. λ_{Church} is just the fragment of $\lambda_{Berkling}$ without protection keys (provided that *var* is infinite). The syntax is then identical, variable occurrences only come as free or bound and there are no free occurrences of a variable x inside any binding context BC_x^n with $n > 0$ (cf. figure 2.2). Consequently, $\Pi_{x,1}^{-1}$ has no effect and $\Pi_{x,0}^{+1}$ does only produce legal λ_{Church}-terms if applied to argument terms without free occurrences of the variable x. This, in turn, restricts the legal substitutions to those of the λ_{Church}-calculus (cf. figure 2.4). Finally, α-conversion is valid only if there are no free occurrences of the new variable in the abstraction body, β-reduction is valid only if there are no free occurrences of variables in the argument that would have to be substituted into abstractions with the same variable name (the substitution rules corresponding to these cases have been invalidated by the restriction to the λ_{Church}-fragment), and η-conversion is valid only if the new variable does not occur free in the term (cf. figure 2.5). To make up for the restricted substitution rules, explicit α-conversions to fresh variables may be necessary before a β-reduction may be performed, which is why an infinite set of variables is needed.

   This proves the first direction of the double implication. The other direction is a consequence of the second part of the correspondence and of the fact that $\forall M_C, N_C \in term_C : M_C =_{\alpha_C} N_C \Leftrightarrow M_C =_{\alpha_B} N_C$. Note, however, that the α_C-convertible terms $(\lambda x.\lambda y.x\ y)$ and $(\lambda x.\lambda z.x\ y)$ are both β_B-reducible, whereas only the latter is β_C-reducible.

2. Any $M_B \in term_B$ without variable occurrences that are protected in M_B is α_B-convertible to a term $M_C \in term_C$ without protection keys (for each abstraction, just choose a variable-name different from any other name occurring free or bound in M_C). Given such $M_B, N_B \in term_B$ with $M_B =_B N_B$, with a proof of equivalence in $\lambda_{Berkling}$ using α_B, β_B and η_B, choose $M_C, N_C \in term_C$ with $M_C =_{\alpha_B} M_B$ and $N_C =_{\alpha_B} N_B$, and imitate the proof using $\alpha_C, \beta_C, \eta_C$. The only complication is that additional α_C-conversions may be necessary to establish the preconditions for β_C- and η_C-conversions (see above).

So, every sequence of $\alpha\beta\eta$-conversions in λ_{Church} is also valid in $\lambda_{Berkling}$. Furthermore, if $\lambda_{Berkling}$-terms without protected occurrences of variables are

$\alpha\beta\eta$-equivalent (with respect to the extended conversion rules), there exist α-equivalent λ_{Church}-terms, which are so, too (with respect to the restricted rules and some additional α-conversions). In other words, given the same infinite set of variables, both calculi allow the same conversions modulo the extended α-equivalence, $\lambda_{Berkling}$ only has more representatives in the α-equivalence-classes, allowing binding structures to be expressed without restrictions on the variable names.

For terms with protected occurrences of variables, a correspondence is not as straightforwardly established because protected occurrences seem to impose more structure on possible contexts which could bind these occurrences. For instance, the term $(\backslash x\ \backslash^2 x)$ requires contexts with at least three λ-abstractions in order to bind both occurrences of x. Moreover, the innermost abstraction will be ignored, and $\backslash x$ and $\backslash^2 x$ will be bound to the second and third innermost abstractions for x, respectively. In contrast, for (ab), both $\lambda a.\lambda b.(ab)$ and $\lambda b.\lambda a.(ab)$ will do, among infinitely many others. Using these properties, it it fairly easy to construct theorems which hold only in one of both calculi. For instance, let

$$fv(M) =_{def} \{v \in var \mid v \text{ occurs free in } M\},$$

then

$$\forall M \in term, x \in var : fv(\lambda x.M) = fv(M) \Leftrightarrow \{x\}$$

holds in λ_{Church}, but not in $\lambda_{Berkling}$ (note that simply including protected occurrences in $fv(M)$ will not suffice here: the theorem has to be adapted, either by restricting $term$ to $term_C$ or by taking protected variable occurrences into account). Based on similar assumptions, equivalences between terms can be proven in λ_{Church} that are not valid in $\lambda_{Berkling}$:

$$\begin{aligned} \forall M \in term: \quad & fv(M) = \{\} \Rightarrow \\ & \forall N, P \in term, x \in var : ((\lambda x.\lambda x.M\ N)P) = (\lambda x.M\ N) = M \end{aligned}$$

(in $\lambda_{Berkling}$, take $M = \backslash x$). However, most of the theoretical work on λ-calculi identifies α-equivalent terms (cf. [Bar84, Appendix C]), so that the most important results carry over to the extended calculi with no or only trivial modifications, which strengthens the point that the conceptual substance of the calculi is not affected by the extensions. If it really becomes necessary to reason about terms that cannot be α-converted to λ_{Church}-terms, it is usually not difficult to generalize the original definitions and propositions so that they hold for the extended calculus and reduce to the known definitions and propositions if only λ_{Church}-terms are involved. This follows from the fact that the extensions are consistent with the original calculus.

The modified representation has definite advantages if the λ-calculus is to be used as the basis of a practical programming language. Firstly, the variable names

$\forall n \in Nat:$

$$\begin{aligned} term_{NF} &= n \\ &\mid \Lambda.term_{NF} \\ &\mid (term_{NF}\ term_{NF}) \end{aligned}$$

$\forall M, N, P \in term_{NF},\ n, k, j \in Nat,\ m \in Int:$

$$\begin{aligned} \Pi_n^m\ k &=_\Pi (k+m),\ if\ k \geq n \\ \Pi_n^m\ k &=_\Pi k,\ if\ k < n \\ \Pi_n^m\ (MN) &=_\Pi (\Pi_n^m\ M\ \Pi_n^m\ N) \\ \Pi_n^m\ \Lambda.M &=_\Pi \Lambda.\Pi_{n+1}^m\ M \end{aligned}$$

$$\begin{aligned} [k \leftarrow N]\ k &=_\sigma N \\ [k \leftarrow N]\ j &=_\sigma j\ ,\ if\ j \neq k \\ [k \leftarrow N]\ (MP) &=_\sigma ([k \leftarrow N]M\ [k \leftarrow N]P) \\ [k \leftarrow N]\ \Lambda.M &=_\sigma \Lambda.\ [k+1 \leftarrow \Pi_0^{+1}\ N]M \end{aligned}$$

$$\begin{aligned} (\Lambda.M\ N) &=_\beta \Pi_1^{-1}\ [0 \leftarrow \Pi_0^{+1}\ N]M \\ M &=_\eta \Lambda.(\Pi_0^{+1} M\ 0) \end{aligned}$$

Figure 2.6: A name-free λ-calculus λ_{NF}

chosen by programmers are never changed during β-reductions, which is important because results modulo α-equivalence are adequate for an abstract theory, but not for real applications. Secondly, it also has advantages for implementations of the transformation system, because all rules are applicable independent of the existence of free or protected variables in the terms to be transformed, in particular, no α-conversions are required prior to β-reductions in order to avoid naming conflicts. Moreover, the conversion rules do even work if the set of variables is finite: all (bound) variables can be renamed by α-conversion without changing variable names but only protection keys. The binding structure is then represented solely by the number of protection keys preceding each variable occurrence, and the variable name, carrying no further information, may even be omitted. This variation, called minimal representation in [Ber76] and summarized in figure 2.6, is isomorphic to de Bruijn's λ-calculus notation with nameless dummies [dB72]. Human readers may have difficulties using this notation for real examples, but the advantages for abstract or mechanized transformations are obvious: the system is greatly simplified by providing exactly one representative for

each class of α-equivalent terms (hence no α-conversion-rule here), and names, name comparisons and renaming are completely replaced by numbers (*de Bruijn indices*) and simple arithmetic. To sum it up, $\lambda_{Berkling}$ embodies both λ_{Church} and $\lambda_{deBruijn}$ as sub-calculi, inheriting both a rich theory and the best prerequisites for an efficient implementation. Furthermore, the combination of variable names with indices allows for a more comfortable user-interface than could be provided using any of the two sub-calculi.

$\forall \mathcal{FORM} : \forall n \in Nat :$

$$\begin{aligned} caf_{\mathcal{FORM}} &= \backslash^n var \mid (caf_{\mathcal{FORM}}\ \mathcal{FORM}) \\ nf &= \lambda var.nf \mid caf_{nf} \\ hnf &= \lambda var.hnf \mid caf_{term} \\ whnf &= \lambda var.term \mid caf_{term} \end{aligned}$$

$$\begin{aligned} c &= [\,] \mid \lambda var.c \mid (c\ term) \mid (term\ c) \\ noc &= \lambda var.noc \mid anoc \\ anoc &= [\,] \mid (anoc\ term) \mid (caf_{nf}\ noc) \\ aoc &= [\,] \mid \lambda var.aoc \mid (term\ aoc) \mid (anoc\ term) \end{aligned}$$

$\forall M, N \in term,\ \lambda x.P, Q \in nf :$

$$\begin{aligned} M \to_\beta N &\Rightarrow c[M] \mapsto_\beta c[N] \\ M \to_\beta N &\Rightarrow noc[M] \mapsto_{\beta,no} noc[N] \\ (\lambda x.P\ Q) \to_\beta N &\Rightarrow aoc[(\lambda x.P\ Q)] \mapsto_{\beta,ao} aoc[N] \end{aligned}$$

Figure 2.7: Normal forms, reduction contexts and reduction strategies

In view of the above, λ shall be used as a synonym for $\lambda_{Berkling}$ throughout the rest of this thesis. Some of the most important results for λ are included in this section for easy reference. For proofs and for more detailed expositions, the reader is referred to [Bar84, HS86]. Figure 2.7 summarizes some notions relevant to the following discussion of reduction. *Normal forms* (nf) are just the irreducible objects of β, *constant applicative forms* (caf) are a useful subset of λ-terms, starting either with a variable or with an application that is not a *redex* (*reducible expression*). *Head normal forms* (hnf) and *weak head normal forms* ($whnf$) are approximations of normal forms ($nf \subset hnf \subset whnf$) that are needed to give consistent interpretations of λ and for practical implementations, respectively.

So far, all conversion rules have been presented as context-free transformation rules in the sense of section 2.1, i.e., they apply in all contexts c. Thus, β

corresponds to an infinite number of rules if the meta-variables for contexts and subterms are instantiated. Whereas the instantiation of subterms is uniquely determined by the redex in question, the instantiation of the context meta-variable is usually possible in more than one way, corresponding to a choice for the next redex to be reduced by the context-free rule. Fortunately, β is a confluent transformation system, so that the order in which redices are selected for reduction (*reduction strategy*) is irrelevant for the result (apart from termination issues).

Theorem 2.2 (Church-Rosser)

1. $\forall M, N_1, N_2 \in term :$
   $(M \rightarrow_\beta^* N_1) \wedge (M \rightarrow_\beta^* N_2) \Leftrightarrow \exists R \in term : (N_1 \rightarrow_\beta^* R) \wedge (N_2 \rightarrow_\beta^* R)$
2. $\forall M, N \in term : M =_\beta N \Rightarrow \exists R \in term : (M \rightarrow_\beta^* R) \wedge (N \rightarrow_\beta^* R)$

Corollary 2.1 (Uniqueness of normal forms)
$\forall M, R_1, R_2 \in term : (M \downarrow_\beta R_1) \wedge (M \downarrow_\beta R_2) \Rightarrow R_1 =_\alpha R_2$

However, β is not a terminating transformation system, i.e., there are reduction sequences of infinite length (even for terms that have a normal form), so that not all reduction strategies are *complete* (they may miss a result while following an infinite path of reductions). The normalization theorem (also known as standardization theorem) states that there is at least one complete reduction strategy, namely normal order reduction ($\mapsto_{\beta,no}$), which is specified in figure 2.7 as β-reduction in normal order contexts (*noc*).

Theorem 2.3 (Normalization) $\forall M, R \in term : M \downarrow_\beta R \Leftrightarrow M \downarrow_{\beta,no}^* R$

Normal order reduction prefers leftmost outermost redices, as can be seen from the definition of applicative normal order contexts (*anoc*): the hole can only be in subterms of an application if the operator is not an abstraction (hence the use of caf_{nf} and *anoc* in operator position), and it can only be in the operand if, in addition, the operator is in normal form. In particular, operand terms of redices are substituted for bound variables without prior reduction which may cause redices in the operand term to be copied. If more than one of the copies is eventually reduced, normal order reduction has multiplied the work to be done. Postponing reductions in argument terms of applications pays only if these terms are substituted for variables which do not occur free in the abstraction bodies, or if all copies of these terms are consumed without contributing to the final result. If this cannot happen, as in Church's λI-calculus, the length of the normal order reduction sequence for a term is an upper bound on the length of any of its reductions [CF74, p. 142].

Applicative order reduction ($\mapsto_{\beta,ao}$), also defined in figure 2.7, prefers innermost redices, performing β-reductions only if both operator and operand have

been reduced to normal form. In doing so, it avoids duplication of redices, but may try to reduce argument terms that are not needed. In case any of these terms does not have a normal form, applicative order reduction does not terminate, regardless of whether the overall term has a normal form or not. It is thus complete only for λI-terms, but not for general λK-terms.

In his PhD thesis, Wadsworth develops a variant of normal order reduction that is complete and does not copy redices [Wad71, chapter 4]. Using graph reduction instead of term reduction, he is able to share reductions in argument terms since copies of arguments are represented as pointers to shared subgraphs. Viewed as term reduction strategy, reducing a redex in one copy of an argument simultaneously reduces all copies of the redex. Wadsworth coined the term *call-by-need* for the corresponding method of parameter passing in programming languages. Applicative order and normal order reduction roughly correspond to *call-by-value* and *call-by-name*, respectively. The correspondence is not exact, however, as the reduction strategies do not only describe methods of parameter passing, but also the order in which reductions (corresponding to function calls) are to be carried out. Applicative order reduction only leaves open the choice which of the innermost redices to reduce first while normal order reduction leaves no choices at all (the next redex to be reduced is uniquely determined). Furthermore, both strategies attempt to reduce to normal forms while, for real programming languages, it is more common to stop whenever a weak head normal form has been reached, i.e., not to reduce inside abstraction bodies (corresponding to function definitions).

To overcome these mismatches, variants of the λ-calculus can be defined that correspond exactly to existing programming languages. For instance, [Plo75] defines a call-by-value calculus λ_V to show that Landin's ISWIM as mechanized by the SECD-machine [Lan63, Lan66]

> *'... is more than a specification of some characterless reduction sequence. Rather, as well as computationally natural, it gives rise to an interesting calculus. Its correspondence with this calculus shows it to be less order of reduction dependent than its definition shows.'*

Plotkin also defines a call-by-name calculus λ_N, for which an appropriate version of the SECD-machine can be defined, and shows how to simulate call-by-name by call-by-value and vice versa. The important difference between λ and λ_V is that β does not hold in λ_V. Instead, only a restricted form β_V holds, where the operand of the redex has to be in $whnf$. This has immediate consequences for languages based on a call-by-value regime. The choice of a non-complete reduction strategy affects the basic conversion rule of the underlying calculus and thus formal reasoning about programs of the language and the usefulness of λ-abstraction. In terms of reasoning, every application of β_V or of the corresponding substitution operator involves a proof that the operand expression in question has a weak head normal form reachable by β_V-reduction. λ-abstraction is affected

because it is not allowed to abstract over a subterm if it cannot be proved that every instance of the subterm has a $\beta_V \Leftrightarrow whnf$, which is usually impossible if the subterm has free occurrences of variables.

2.3 From λ-calculi to functional languages

The terms and reduction rules of the λ-calculus do already form a simple but computationally complete programming language. Just as in the early days of programming, programmers could map problem specifications to λ-terms, have them evaluated according to the reduction rules and interpret normal forms as problem solutions. That such a scheme would work was first formulated by Church and has become known as Church's thesis ('the notion of an effectively calculable function of positive integers *[can]* be given an exact definition by identifying it with that of a λ-definable function' [Chu51, p. 41]). This general thesis cannot be proven, but is a plausible attempt to generalize other, formally proven results: the functions on integers definable in λ coincide with the partial recursive functions, and λ-calculus has also been shown to be computationally equivalent to other notions of computation, e.g., Turing-machines.

While theoretically enlightening, this is not an adequate approach to practical programming as the mapping between objects in the problem domain and those of the λ-calculus and back is too expensive. To devise a practical programming language based on λ-calculus requires that either the language is equipped with means to handle objects of the problem domain directly or, at least, that it provides support for the mappings in order to reduce the efforts needed to represent these objects in terms of the programming language. In other words, having established a system of sufficient computational power, it needs to be extended with respect to expressive power. When comparing computationally equivalent languages, a language will be called more expressive for a given problem domain if the representation of objects of the domain in terms of objects of the language is simpler.

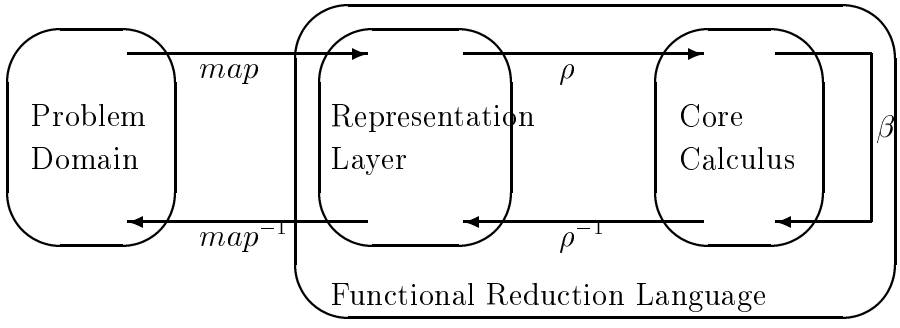

Figure 2.8: Enhancing expressiveness by adding a representation layer

The basic solution (cf. figure 2.8) is to embed the λ-calculus into a larger system that maps high-level representations into λ-terms (ρ), reduces them, and maps the results back into high-level representations (ρ^{-1}), thus reducing the effort that comes with the user-supplied mappings (map, map^{-1}). However, in order to assure that the mapping is bijective, it is necessary to keep additional information, a kind of labels, in the low-level representation. These labels are irrelevant to reduction and are only used when mapping results back into high-level representations. In other words, low-level representations can be factorized into *labels* and *computational contents*, and terms that differ only in their labels are equivalent as far as reduction is concerned. The calculus has to be extended to work on equivalence classes of terms, leaving labels unchanged.

As a simple example, consider α-equivalence: the computational content of a λ-abstraction is given by its binding structure, i.e., its influence on variables of the same name occurring free or protected in its body. The name itself is only a label, irrelevant to reduction, as shown by the possible translation into λ_{NF}. While, in the original λ-calculus, it may be necessary to change the names to perform further β-reductions, thereby loosing any chance to present (intermediary) reduction results with the same names as chosen in the start term, the names are left unchanged in the extended λ-calculus. Translation into λ_{NF}-terms will usually be part of ρ but is omitted here to enhance legibility. Therefore, λ-terms are not modified by the mapping and are left unlabeled.

Further examples of representation mapping include the omission of parentheses and λs assuming association to the left and to the right, respectively, and the introduction of syntactic sugar for (recursive) definitions, numbers, booleans, conditional expressions, data structures, etc., some of which are sketched in figure 2.9. The additional symbols $\lfloor$, $\rfloor$ and the labels will in most cases be omitted from now on, which means they must be maintained through all conversions. Church used a similar system of abbreviations in [Chu51][1], but did only formalize the direction from left to right (nominal and schematic definitions). He used his human intuition to introduce or keep the right abbreviations throughout reduction sequences, whereas the labels and representation mappings used here allow this to be mechanized. Without the labels, it could not be decided whether $(\lambda x.M\ N)$ should be retranslated into a *let*-construct or left as it is, and $\lambda xy.y$ could stand for $zero$, for $false$, or for $\lambda x.\lambda y.y$. Both the left-to-right and the right-to-left transformation system are terminating and can be kept confluent due to the use of labels: they define a representation mapping ρ and its inverse ρ^{-1}. Note that labels are attached mostly to abstractions or terms in operator position. If a β-reduction changes the shape of the defining term in such a way that the retranslation step corresponding to the label becomes impossible, it will also consume the label together with the operator.

Using (refined) representation mappings similar to the ones given here, it

[1] cf. also [Bar96b] for further structure-preserving encodings of data types as λ-terms.

$\forall M, M_1, M_2, N, P \in term,\ \vec{M} \in term^*,\ v \in var,\ \vec{v} \in var^+,\ k, i \in Nat$:
Let $p, f, g, x, y, n, l, h, t \in var$ (pairwise different). Then

$$
\begin{array}{rcl}
Y & =_\rho & \lfloor \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x))) \rfloor_Y \\
I & =_\rho & \lfloor \lambda x.x \rfloor_I \\
K & =_\rho & \lfloor \lambda xy.x \rfloor_K \\
(M \circ N) & =_\rho & \lfloor \lambda x.(\Pi^{+1}_{x,0} M\ (\Pi^{+1}_{x,0} N\ x)) \rfloor_\circ \\
M^0 & =_\rho & I \\
M^k & =_\rho & \lfloor M^{k-1} \circ M \rfloor_{repeat},\ if\ (k > 0) \\
(M_1\ M_2\ \vec{M}) & =_\rho & ((\lfloor M_1 \rfloor_{()}\ M_2)\ \vec{M}) \\
\lambda v\ \vec{v}.M & =_\rho & \lfloor \lambda v.\lambda \vec{v}.M \rfloor_\lambda \\
let\ v\ =\ N\ in\ M & =_\rho & (\lfloor \lambda v.M \rfloor_{let}\ N) \\
let\ f\ \vec{v}\ =\ N\ in\ M & =_\rho & let\ \lfloor f \rfloor_{fundef}\ =\ \lambda \vec{v}.N\ in\ M \\
letrec\ v\ =\ N\ in\ M & \to_\rho & let\ v\ =\ \lfloor (Y\ \lambda v.\Pi^{+1}_{v,1} N)\ \rfloor_{letrec}\ in\ M \\
letrec\ v\ =\ N\ in\ v & \leftarrow_\rho & \lfloor (Y\ \lambda v.\Pi^{+1}_{v,1} N)\ \rfloor_{letrec} \\
letrec\ f\ \vec{v}\ =\ N\ in\ M & =_\rho & letrec\ \lfloor f\ \rfloor_{fundef}\ =\ \lambda \vec{v}.N\ in\ M \\
if\ M\ then\ N\ else\ P & =_\rho & (\lfloor M \rfloor_{if}\ N\ P) \\
true & =_\rho & \lfloor \lambda xy.x \rfloor_{true} \\
false & =_\rho & \lfloor \lambda xy.y \rfloor_{false} \\
\langle M, N \rangle & =_\rho & \lfloor \lambda f.(f\ \Pi^{+1}_{f,0} M\ \Pi^{+1}_{f,0} N) \rfloor_{Pair} \\
fst & =_\rho & \lfloor \lambda p.(p\ \lambda xy.x) \rfloor_{fst} \\
snd & =_\rho & \lfloor \lambda p.(p\ \lambda xy.y) \rfloor_{snd} \\
\langle \vec{M} \rangle & =_\rho & \lfloor \lambda f.(f\ \ \overrightarrow{\Pi^{+1}_{f,0}\ M}) \rfloor_{\langle\rangle} \\
select_{i,k} & =_\rho & \lfloor \lambda t.(t\ (K^i\ K^{(k-i-1)})) \rfloor_{select_{i,k}} \\
extend & =_\rho & \lfloor \lambda xy.(y \circ x) \rfloor_{extend} \\
zero & =_\rho & \lfloor \lambda fx.x \rfloor_{zero} \\
succ & =_\rho & \lfloor \lambda n.\lambda fx.(f\ (n\ f\ x)) \rfloor_{succ} \\
\underline{k} & =_\rho & \lfloor (succ^k\ zero) \rfloor_{num} \\
add & =_\rho & \lfloor \lambda nm.\lambda fx.(n\ f\ (m\ f\ x)) \rfloor_{add} \\
mult & =_\rho & \lfloor \lambda nm.(n \circ m) \rfloor_{mult} \\
nil & =_\rho & \lfloor \lambda fy.y \rfloor_{nil} \\
cons & =_\rho & \lfloor \lambda ht.\lambda fy.(f\ h\ (t\ f\ y)) \rfloor_{cons} \\
isempty & =_\rho & \lfloor \lambda l.(l\ \lambda ht.false\ true) \rfloor_{isempty} \\
fold & =_\rho & \lfloor \lambda fxl.(l\ f\ x) \rfloor_{fold}
\end{array}
$$

Figure 2.9: Examples of representation mappings

is possible to turn the λ-calculus into an expressive functional language. However, depending on the target architecture, there are usually more direct ways to represent, e.g., numbers and data structures or to implement recursion. Furthermore, since the representation layer is implemented by means of representation-independent reduction rules, stepwise reductions may proceed through intermediate terms that have no counterpart in the representation, and the interactions between labeled terms are not restricted to those natural for their representations. The latter is not necessarily a bad thing, e.g., applying the Church numeral $\underline{n}$ to a function yields the n-fold repetition of that function, and multiplying two functions yields their composition, but, in general, it is preferable to have reduction rules that correspond with the level of representation. This is the way our reduction language will be defined in chapter 5.

Introducing such representation-level reduction rules requires great care as the resulting reduction language should be a consistent extension of the core calculus. One way to guarantee this is to make sure that the extensions could be consistently modeled by a representation mapping. Each new rule is then only an abbreviation for a sequence of steps in $\rho^{-1^*} \circ \beta^* \circ \rho^*$, abstracting from the lower-level details of representation mapping and intermediate steps and introducing larger reduction steps only for terms which are already convertible. Such an analogy also helps to clarify the nature of new binding constructs ($let, letrec$) and provides a guideline on how to extend the substitution and conversion rules to deal with the additional language constructs, which is indispensable in order to carry over the properties of the core calculus to the representation layer.

2.4 Summary

Starting from a fairly general model of programming, the major design decisions leading to reduction languages have been outlined, and a brief review of their formal basis has been given. Many of these first design decisions led to restrictions, which seems to be against the spirit of our design principles. These restrictions have been justified by the observation that transformation systems, while being simple, general and expressive, also make it very easy to specify systems that do not exhibit any of the useful properties we have defined. In general, it is even undecidable whether or not a given system is, e.g., confluent. This is too much flexibility for almost all uses of a programming language, and so we have been searching for a compromise that allows to guarantee useful properties – and thus formal reasoning – without giving up too much of the flexibility.

Some of the restrictions, e.g., the exclusion of non-confluent systems, were made in order to simplify the language framework and should not be taken as final decisions. Since non-determinism is a useful abstraction mechanism (it can be viewed as abstraction over unknown or irrelevant parameters, i.e., parameters which can only be guessed or which may be chosen randomly), it remains to be

seen how far we can get without it, and whether it is possible to extend functional languages with non-deterministic features without destroying their characteristic properties. Other restrictions, namely those following from the use of λ-calculi and reduction languages as an intermediate level between transformation systems and programmers, were necessary to provide programmers with a less fragile framework and were compensated by gains in program structure (local definitions), expressiveness (programs as data) and reasoning capability.

Reduction languages are consistent extensions of λ-calculi and are defined by confluent systems of context-free transformation rules. Therefore, they allow equational reasoning with respect to the equational theory induced by the reduction rules. This means that programs which can be proven equal in the theory can be replaced by each other in all contexts (this property is often called *referential transparency* [SS90]). Indeed, the whole execution model is based on equivalence-preserving program transformations (reductions) and thus completely independent of particular machine architectures. The Church-Rosser property (or confluence) of the transformation system guarantees that the choice of reduction strategy has no influence on the results of computations (apart from termination), which means that the strategy can be adapted to operational requirements. If sufficient resources are available, independent reductions can be performed non-sequentially (in any order). Furthermore, there are complete reduction strategies that are guaranteed to find equivalent normal forms for programs which have one.

Substitution revisited

β-conversion, based on the definition of substitution given in figure 2.4, looks rather complicated. Furthermore, viewed as a transformation system, it is not efficient: the abstraction body is traversed twice, and the operand term is traversed at least once before the substitution to protect free occurrences of variables against the additional binding and then once per instance to unprotect the occurrences again when the λx is removed. While the specification is modular, it shows too much detail: first, free occurrences of x are substituted by copies of the operand term (with additional protection), then the λx is removed, and one level of protection is removed from protected occurrences of variables in the former abstraction body.

As β is the main conversion rule of the λ-calculus, it seems worthwhile to simplify the rule, using information about the combination in which Π and substitution are used. In figure 2.10, a modified substitution is defined that incorporates the functionality of Π needed for the specification of β. A simplified definition of this substitution is then formally derived by an analysis of cases. Viewed as a transformation system, it traverses the abstraction body only once and avoids the superfluous protection and unprotection traversals of the operand term. The modified definitions have been used, e.g., in [BF82], and the corresponding name-free rules are also closer in spirit to the rules used in [dB72].

$\forall M, N \in term,\ x, y \in var,\ k, j, n \in Nat$:

$$[\backslash^k x \leftarrow N]^{-1}_{x,n}\ M \quad =_{def} \quad \Pi^{-1}_{x,n+1}\ [\backslash^k x \leftarrow \Pi^{+1}_{x,n}\ N]\ M$$

$$\begin{aligned}
&[\backslash^k x \leftarrow N]^{-1}_{x,n}\ \backslash^k x \\
&\quad =_{\sigma} \quad \Pi^{-1}_{x,n+1}\ \Pi^{+1}_{x,n}\ N \\
&\quad =_{\Pi} \quad N \\
&[\backslash^k x \leftarrow N]^{-1}_{x,n}\ \backslash^j x\ ,\ if\ (j > n)\ \wedge (j \neq k) \\
&\quad =_{\sigma} \quad \Pi^{-1}_{x,n+1}\ \backslash^j x \\
&\quad =_{\Pi} \quad \backslash^{j-1} x \\
&[\backslash^k x \leftarrow N]^{-1}_{x,n}\ \backslash^j y\ ,\ if\ ((j < n) \wedge (j \neq k)) \vee (y \neq x) \\
&\quad =_{\sigma} \quad \Pi^{-1}_{x,n+1}\ \backslash^j y \\
&\quad =_{\Pi} \quad \backslash^j y \\
&[\backslash^k x \leftarrow N]^{-1}_{x,n}\ (M\ N) \\
&\quad =_{\Pi,\sigma,def} \quad ([\backslash^k x \leftarrow N]^{-1}_{x,n}\ M\ [\backslash^k x \leftarrow N]^{-1}_{x,n}\ N) \\
&[\backslash^k x \leftarrow N]^{-1}_{x,n}\ \lambda x.M \\
&\quad =_{\Pi,\sigma} \quad \lambda x.\Pi^{-1}_{x,n+2}\ [\backslash^{k+1} x \leftarrow \Pi^{+1}_{x,0}\ \Pi^{+1}_{x,n}\ N]\ M \\
&\quad =_{\Pi} \quad \lambda x.\Pi^{-1}_{x,n+2}\ [\backslash^{k+1} x \leftarrow \Pi^{+1}_{x,n+1}\ \Pi^{+1}_{x,0}\ N]\ M \\
&\quad =_{def} \quad \lambda x.[\backslash^{k+1} x \leftarrow \Pi^{+1}_{x,0}\ N]^{-1}_{x,n+1}\ M \\
&[\backslash^k x \leftarrow N]^{-1}_{x,n}\ \lambda y.M\ ,\ if\ y \neq x \\
&\quad =_{\Pi,\sigma} \quad \lambda y.\Pi^{-1}_{x,n+1}\ [\backslash^k x \leftarrow \Pi^{+1}_{y,0}\ \Pi^{+1}_{x,n}\ N]\ M \\
&\quad =_{\Pi} \quad \lambda y.\Pi^{-1}_{x,n+1}\ [\backslash^k x \leftarrow \Pi^{+1}_{x,n}\ \Pi^{+1}_{y,0}\ N]\ M \\
&\quad =_{def} \quad \lambda y.[\backslash^k x \leftarrow \Pi^{+1}_{y,0}\ N]^{-1}_{x,n}\ M
\end{aligned}$$

$$\begin{aligned}
\lambda x.M \quad &=_{\alpha} \quad \lambda y.[x \leftarrow y]^{-1}_{x,0}\ \Pi^{+1}_{y,0}\ M \\
(\lambda x.M\ N) \quad &=_{\beta} \quad [x \leftarrow N]^{-1}_{x,0}\ M
\end{aligned}$$

Figure 2.10: Alternative definitions of substitution, α- and β-conversion

This example serves to demonstrate two points. Firstly, both functional programs and the fundamental rules of the calculi they are based on are suited for formal reasoning. The style of reasoning is a straightforward equational one, replacing equals by equals according to the equivalence relations induced by the transformation systems. Secondly, since transformation systems directly support both declarative and operational views of specifications, it is a common misconception to see the purely operational view as the only possible implementation of a given specification, leading to imprecise statements such as 'substitution is inefficient because it traverses the abstraction body twice'. As the example derivation has shown, such operational attributes may be used with respect to operational views of a given specification, but the declarative view of the very same specification usually allows various possibilities of implementation, related by formal derivations. The operational view guarantees that a given specification can be implemented at all, which allows rapid prototyping and supports the idea of 'getting it right before getting it fast', but to find efficient implementations is usually more difficult. Some of these optimizations may have to be performed by application programmers, some of them may be built into optimizing compilers, and some may be performed by system programmers when upgrading the language implementation. However, due to the decision to base reduction languages on calculi defined by transformation systems with certain properties, all these activities have a formal basis which is rather straightforward to use.

Chapter 3

Input/Output and State in Functional Languages

There have been many seemingly different approaches to the problem of doing input/output from within a functional programming language. Over the last decade, the emphasis has shifted (at least in the research community) towards adequate solutions for purely functional languages, i.e., towards extensions for input/output that do not compromise the properties of these languages. In this chapter, we review some of the major variants, but we do so not in a chronological order or as an attempt to compare instances of input/output-schemes in popular functional languages. Instead, we try to outline a logical development that is behind the various approaches.

We start from a very naive problem specification and obvious solutions. For each proposed solution, a discussion of its major pros and cons leads to possible enhancements either of the solution or of the problem specification. Following some of the possible lines of development, we encounter basic forms of all popular input/output-schemes. We identify the modifications that distinguish schemes lying on the same line of development or the design decisions that separate schemes on different lines. We also investigate the close relationship between the treatment of input/output and global states.

Though the form of presentation is new, most of the concepts and approaches to input/output in functional languages have been developed and used earlier. To put things into the right perspective, the section closes with a brief overview of related work and surveys on the topic. The necessary references are given there, too.

A note on the program notation

In addition to transformation systems, the discussion of different approaches to input/output uses a small number of example problems, for which programs are given in each of the input/output-styles discussed here. A Haskell-like notation

[PH96] is used to provide a uniform notational basis for all examples. Note that, even though several input/output-styles have been used in the development of Haskell, none of the programs in this chapter is likely to be a valid Haskell program with the intended meaning. The following major constructs are used: mutually recursive definitions (listed between the keywords `let` and `in`, valid in the expression following the keyword `in`), expression lists (in square brackets `[ ]`), also explicitly used as binary lists (*head*:*list*), tuples (in round brackets `( )`), λ-abstractions (written as `\`*var*`->`*expr*), and function applications (in round brackets `( )`). Application associates to the left, and bodies of λ-abstractions extend as far as possible to the right, allowing many round brackets to be omitted. Pattern-matching may be used in definitions or in λ-abstractions. We also provide type annotations for documentation (*expr* `::` *type*), and data type definitions (keyword `data`). Keywords and variable names start with lower-case letters, whereas data constructors and type names start with upper-case letters. The keywords `let` and `in` are omitted for top-level definitions, and the identifier `main` defines the start expression of a program. Further constructs and auxiliary functions are described where they are used.

Furthermore, for some input/output-styles, non-strict evaluation is a necessary prerequisite, so we define the notion of strictness here. For a given transformation system $\rightarrow_T$ over expressions $expr$, a context $C \in C_{expr}$ is called *strict*, if the evaluation of $C[\ e\ ]$ depends on the evaluation of e, for all $e \in expr$:

$$C \text{ strict} \Leftrightarrow_{def} \forall e \in expr : C[\ e\ ]\downarrow_T \Rightarrow e\downarrow_T$$

In *non-strict* functional languages, the strictness of the context `(`f `[ ])` depends only on the function f (so the strictness can also be attributed to the function), whereas in strict languages, this context is always strict (for instance, due to a call-by-value regime). Note that this definition of strictness of a programming language is based on its general treatment of application only. Even in strict languages, there are usually some non-strict contexts (alternatives of if-then-else, bodies of abstractions, right hand sides of recursive definitions), and applications of, e.g., arithmetic primitives are strict even in non-strict languages.

3.1 The very first idea: using side-effecting pseudo-functions

On first sight, it might be tempting to simply copy the input/output-operations of existing imperative languages, based on a first specification of the problem:

> *functional languages should provide input/output-facilities similar in expressive power to those of imperative languages.*

This idea, if implemented naively, leads to quite a number of problems, and even if these problems are avoided by a careful redefinition of the language semantics, the result is far from satisfying. To illustrate this, we assume that we have an operation each to get a line of text from the keyboard and to put a string to the screen:

```
gets :: String              -- input a line of text, return text
puts :: String -> String    -- output a string, return parameter
```

Both `gets` and `puts`, in addition to returning a string, interact with the program's runtime environment: `gets` fetches a new line of text from an input device each time it is called and returns the text as a string of characters. `puts` just returns its parameter, but also delivers the string to an output device. We call these operations *pseudo-functions* because they have side-effects but are treated as if they were primitive functions.

The question then becomes: When (in terms of relative ordering) and how are these operations to be carried out, and what are their results or what do they effect? The natural answer to the first part seems to be: In the order in which the operations appear in the program text. A list of operations would then correspond to a sequence of interactions performed from left to right:

```
[ puts "what's your name?", gets, puts "are you sure?\n", gets]
```

But how can the input data be passed on to parts of the program that occur after the call to `gets` (both in the text and in the sequence of events), e.g., how can the string returned by `gets` be used in the second call to `puts`? And what about two syntactically equivalent expressions (`gets`) giving different results? Take, for example, the following piece of program:

```
let x = gets
in f x x
```

Is this intended to read one line of input and substitute it for both parameters of `f`? Or should `x` be substituted by the operation `gets` itself, resulting in two input operations with possibly different results? Also, the seemingly simple 'in the order of occurrence'–rule is hardly adequate in the presence of β-reduction, as in the following small program:

```
let
  f x y = g y x
  g x y = h y y
  h x y = "what?"
in f (puts "hello ") (puts "world\n")
```

What is this program supposed to write on the screen? Depending on the interleaving of reductions and input/output-operations, it might print any of `"hello world\n"`, or `"world\nhello "`, only `"hello "`, only `"world\n"`, twice `"hello "`, or nothing at all, since none of the parameters needs to be evaluated to compute the final result of `"what?"`. It is all too clear that we need a better understanding of the problem and of the input/output-facilities of imperative languages before we can hope to come up with proposals that are more adequate for functional languages.

3.2 What is input/output?

Roughly speaking, input/output is a general term that subsumes all kinds of interactions between a running program and its runtime environment. This includes terminal input/output, but also file operations and communication with external devices (printers, ...) and with other programs, possibly running on foreign systems (networking). For simplicity, the following discussion is restricted to character-based terminal-input/output. To understand why input/output-operations cannot simply be copied from imperative languages, where they seem to cause no problems at all, to functional languages, it is necessary to compare the programming paradigms. Both paradigms are easily represented as different kinds of transformation systems, providing a common basis for further discussions.

Imperative programming is all about program-controlled sequences of state transformations. Leaving all the details aside, programs are executed by an abstract machine that has access to the program (a sequence of abstract machine instructions), a program counter (the address of an instruction in the program), and a store of box variables (for procedural languages, the store will also include a stack to hold procedure parameters, temporary values and return addresses). The essence of this paradigm can be represented by a system of transformations of the abstract machine state (using $||$ to separate the program from the rest of the state):

$$(instr_a)_{a \in Addr} \;||\; < pc, store > \;\mapsto\; (instr_a)_{a \in Addr} \;||\; (instr_{pc} \; < pc, store >)$$

A program is represented by a sequence of instructions, indexed by addresses. The sequence of state transformations is totally ordered: the value of the program counter (pc) determines the next instruction to be executed ($instr_{pc}$), which specifies both the transformation of the store and the new value of the program counter.

In a certain sense, imperative programs do nothing but input/output: their only purpose is to interact with the state of the machine they are running on and to specify exactly the transformations of this state. It is a simple task to integrate further input/output-operations into such a system: part of the

store is reserved for this purpose (input/output-registers) and can be accessed by the input/output-devices, i.e., the input-device delivers any inputs in the input-register, and the contents of the output register are processed by the output-device. The access of devices to the registers is assumed to be external to our discussion and is only reflected in the state of the registers (*string* or *char* if a string or character is waiting to be processed and $\Box$ if a register is unused). The purpose of the input/output-operations is then to transfer strings of characters back and forth between the input/output-registers and the store. So, input/output-operations access a larger part of the machine state, but fit into the picture immediately – they can appear anywhere in a sequence of instructions, and their effect is a transformation of the machine state:

$$\begin{aligned}(instr_a)_{a \in Addr} \;||\; &< pc, in, out, store > \\ \mapsto\; &(instr_a)_{a \in Addr} \;||\; (instr_{pc} \;< pc, in, out, store >)\end{aligned}$$

In contrast, functional programming is about program transformations. Programs are executed by an abstract machine that replaces expressions by semantically equivalent ones according to the context-free rules of a reduction calculus:

$$expr \;\rightarrow\; expr`$$

The essential differences to the imperative paradigm are that program transformations are context-free and only partially ordered. While context-free transformation rules greatly simplify the understanding of programs and their evaluation, they also mean that program evaluation is fully independent of the context – neither are the program transformations sensitive to context information nor do they have any effect on the state of a context. Since the sole purpose of input/output is to let a program interact with the context in which it is evaluated, context-sensitive transformations have to be introduced to the functional paradigm. The first attempt was to add rules for primitive input/output-operations (to interact with some global input/output-registers) without changing anything else. In other words, both program transformations and input/output-operations are allowed in all program contexts C (cf. figure 3.1).

$$\begin{aligned}&C[\ expr\ ] \;||\; < in, out > \\ &\quad\mapsto\; C[\ expr`\ ] \;||\; < in, out >,\ if\ expr \rightarrow expr` \\ &C[\ \texttt{gets}\ ] \;||\; < string, out > \\ &\quad\mapsto\; C[\ string\ ] \;||\; < \Box, out > \\ &C[\ (\texttt{puts}\ string)\ ] \;||\; < in, \Box > \\ &\quad\mapsto\; C[\ string\ ] \;||\; < in, string >\end{aligned}$$

Figure 3.1: Input/output with pseudo-functions

The problem with this approach is that the resulting transformation system is no longer confluent: program transformations and interactions are only partially ordered by data dependencies in the program text, and the order in which interactions occur has an influence on the values returned by the pseudo-functions. It is now clear why it is not sufficient to specify a static textual ordering for interactions: every such ordering could be modified by program transformations (in particular, calls to the pseudo-functions may be consumed or duplicated by reduction). The sequence of interactions is thus not fully determined, which not only makes the program output (via `puts`) hard to predict but also affects the results of program transformations by the uncontrolled transfer of objects from the input-register to the program (via `gets`).

In the imperative case, this problem is avoided due to the total ordering of all state transformations: the system state includes a program counter, which uniquely determines the next instruction to be performed. This can be either an input operation, an output operation or some other state transformation, but there is never a choice. Of course, the program transformations in the functional case could also be ordered totally in order to achieve deterministic system behavior in the presence of side-effecting primitive pseudo-functions, but this would be a very high price to pay for input/output. Also, referential transparency and thus the possibility to do equational reasoning independent of the context would be lost. We conclude that side-effecting pseudo-functions do not fit well into the functional framework, and we add the following constraint to our first problem specification:

> *The addition of input/output-facilities should not compromise characteristic properties of functional programming languages such as referential transparency.*

Our problem specification still requires expressive power similar to imperative input/output, but simply copying input/output-primitives from imperative languages would not allow us to meet the additional constraint. So, we may start from scratch and ask: how do we get something into or out of a functional program? The answer seems to be quite simple because this is what functions are about – mapping input to output – but, as we will see, there are still some problems to be solved and design decisions to be made. From now on, we will use a fixed set of very simple problems to investigate the different syntactic constructs of each proposed input/output-system. The example problems are

1. *a simple login procedure:* The user is prompted for his name, enters the name and is greeted by the system. This is intended to illustrate the basic input/output-constructs.

2. *a simplified two-player dialog:* Two players log into a system and enter a dialog, which is not modeled in detail, but simply echoes the names of both

players. This uses the login procedure in a larger context and is intended to illustrate the means for composition of programs involved in input/output.

3.3 Utilizing function parameters and values

A direct way of getting computational objects in and out of a function is to pass them via the function's parameters and via the function value, respectively. The same scheme could be used to embed a functional program (as a function from its input to its output) into an environment (cf. figure 3.2).

$$
\begin{array}{l}
\texttt{start}\ \lfloor \texttt{main} \rfloor\ ||\ < string, out > \\
\qquad \mapsto\ (\texttt{main}\ string)\ ||\ < \Box, out > \\
C[\ expr\ ]\ ||\ < in, out > \\
\qquad \mapsto\ C[\ expr`\ ]\ ||\ < in, out >,\ if\ expr \rightarrow expr` \\
string\ ||\ < in, \Box > \\
\qquad \mapsto\ \texttt{done}\ ||\ < in, string >
\end{array}
$$

Figure 3.2: Functional programs as functions from input to output

Multiple inputs and outputs may be collected in data structures, and since a total order on the inputs and outputs is needed, the natural choice is to pass both inputs and outputs in lists, the functional language equivalent to sequences. The good thing about this approach is that there are no explicit input/output-operations that could be in conflict with each other or with reduction steps. The inputs are implicitly provided on program startup and the outputs are generated implicitly after program termination. The bad thing is that the complete sequence of inputs has to be fixed before the program is executed, and the complete sequence of outputs becomes accessible only after the execution has terminated.

3.3.1 Streams

The idea sketched above seems to be based on the right data dependencies (inputs are supplied as parameters to the program, and outputs are simply the program value), but has inconvenient synchronization properties, because reduction starts only after all inputs are available, and output starts only after reduction has terminated. This is mainly due to overly pessimistic strictness assumptions, assuming that all inputs are needed to start any reductions, and that no outputs can be generated before reduction has terminated. These assumptions are unrealistic: in an interactive environment, it should be possible to add elements to the sequences of inputs and outputs continually, i.e., there should be *streams* of inputs and outputs instead of fixed sequences. For instance, in a typical session

with an interactive program, the program is first started, then produces output to prompt for inputs which are provided only afterwards, and the program continues with some computation based on the input data. This process may be repeated ad infinitum.

Character streams

With non-strict evaluation, and with non-strict list constructors in particular, possibly infinite streams can be represented as partially unevaluated lists. Initial segments of the output list may be produced without full evaluation of the output list (which is the value of the program's main function applied to the list of inputs), and the evaluation of the program's main function may start before its input list has been even partially evaluated, i.e., before any inputs have been made. Refining the parameter/value-idea with non-strict evaluation and characters as input/output-objects leads to a system known as *character streams* (cf. figure 3.3), in which input, output, and reduction can take place in any order as long as the data dependencies are respected. The functional program takes as its parameter a non-strict list of characters (denoted by the special variable `in` here) from an input device and produces as its result a non-strict list of characters for an output device.

$$
\begin{array}{l}
\texttt{start}\ \lfloor \texttt{main} \rfloor\ ||\ < in, out > \\
\qquad \mapsto\ (\texttt{main in})\ ||\ < in, out > \\
C[\ expr\ ]\ ||\ < in, out > \\
\qquad \mapsto\ C[\ expr`\ ]\ ||\ < in, out >,\ if\ expr \rightarrow expr` \\
expr\ ||\ < char, out > \\
\qquad \mapsto\ [\ \texttt{in} \leftarrow (char : \texttt{in})\ ]\ expr\ ||\ < \Box, out > \\
expr\ ||\ < \texttt{eos}, out > \\
\qquad \mapsto\ [\ \texttt{in} \leftarrow [\ ]\ ]\ expr\ ||\ < \Box, out > \\
char : expr\ ||\ < in, \Box > \\
\qquad \mapsto\ expr\ ||\ < in, char > \\
[\ ]\ ||\ < in, \Box > \\
\qquad \mapsto\ \texttt{done}\ ||\ < in, \texttt{eos} >
\end{array}
$$

Figure 3.3: Input/output with character streams

If sufficient resources are available, the transformation rules of figure 3.3 can be applied in any order (even non-sequentially). However, a minimal demand-driven approach to transformation is also possible, in which the demand for the output stream drives the entire evaluation process. The demand for characters in the output stream (fifth rule) causes further reductions (second rule), and if these reductions depend on characters in the input stream, further inputs are requested

from the input device (third rule). In any case, both the input operations and the output operations are totally ordered, but otherwise there is only a partial ordering among reductions and input/output interactions, which is solely determined by data dependencies. Note that reductions may proceed as long as they do not depend on the value of `in`, in particular, `in` may be duplicated or consumed due to reductions (inside the program, `in` is a free variable that represents the not yet available part of the input stream). Therefore, all occurrences of `in` need to be substituted when further input becomes available. Note also that neither the input nor the output stream are ever modified: their values are only refined when further inputs or outputs become available. Stream communication is terminated by special constants `eos` (*end of stream*), which correspond to empty lists (`[ ]`) in the list representation of streams (note the difference to □, which represents a not yet filled register).

```
main input = "What is your name? "
           ++ (hello (hd (lines input)))

hello name = name ++ "\n"
           ++ "Hello, " ++ name ++ "!\n"
```

Figure 3.4: The login example with character-streams

Using character-streams for input/output, the login procedure (figure 3.4) can be described basically as the concatenation of the output strings, which comprise a prompt, the username (to echo the input), and the greeting (which also includes the username). In addition to list and string concatenation (`++`), the following functions are assumed to be predefined: `lines` takes a string of characters and splits it into a list of strings, using the newline-character `'\n'` as a delimiter; `hd` and `tl` return the first element and the rest, respectively, of a binary list. For convenience, the input stream, which is available as the parameter `input` of `main`, is restructured into a list of lines. Only the first line of input (`(hd (lines input))`), which carries the username, is passed on to `hello`. Note how the name is echoed in the output string to make sure that the function value depends on the input, which is therefore requested *before* the `"Hello, "`-part of the greeting is printed. Also, `input` cannot be inspected by pattern matching in `main`, because if the function body depended on the structure of the `input` parameter, input would be requested before the prompt was written.

While `hello` can be directly reused for the dialog example (figure 3.5), a small modification to `login` (the former `main`) is necessary. The conversion of the input stream into a list of lines only needs to be done once, in the new `main`-function, so `login` already gets a list of input lines (`ls`) as its parameter. In addition to

the output string, `login` also returns the name that has been read and all but the first line of the input stream. The latter information is needed because the calling function (`main` in this case) has no other possibility to determine what part of the input stream has been read. The function `main` ties the parts together: `login` is called once for each player, first with the complete list of input lines, and then with the lines left over by the first call. The function `dialog` is called with the lines left over by the second `login` and two usernames as parameters, and the concatenated output of the calls to `login` and `dialog` is returned as the value of `main`.

```
hello name = name ++ "\n" ++ "Hello, " ++ name ++ "!\n"

login ls = let name = hd ls
           in (tl ls, "What is your name? " ++ (hello name), name)

dialog ls a b = "player A: " ++ a ++ " player B: " ++ b ++ "\n"

main input = let
               ls = lines input
               (ls1,out1,a) = login ls
               (ls2,out2,b) = login ls1
             in out1 ++ out2 ++ (dialog ls2 a b)
```

Figure 3.5: The dialog example with character-streams

Even though recursive definitions are given in the sequence in which the functions are called, the sequence of interactions is determined solely by the data dependencies (the value of main depends on `out1`, which depends on the value of `(login ls)`, etc.) and the demand for the output of `main`. The composition of program parts involved in input/output is a tedious task and best hidden inside a general composition function. However, to make this work, all subprograms have to use a common interface, e.g., take the list of input lines as parameter and return as function value a tuple composed of the unread input, the output and some return value. Note that the only function in the example using this interface is `login`. The other functions, including those mapping an input stream to an output stream, could not be be composed this way.

Character streams can be quite convenient for pure text processing, but the style has a number of drawbacks: it is not very flexible, as there is no simple way to redirect input or output to other but the predefined devices or to communicate something else but strings of characters, e.g., to indicate status conditions. The way in which interactions and reductions are interleaved requires the use of a

demand driven reduction strategy. This, in turn, makes the order of events sometimes hard to predict, especially the synchronization of input and output operations. And last, but not least, functions mapping input to output streams are not directly composable, so that the basic element of this input/output-style, functions from streams to streams, cannot be the basic building block for program construction.

Request-/Response streams

The major causes of the low flexibility of character-streams are that only the objects of interaction (characters) are embedded into the functional programs, whereas interactions themselves happen implicitly by mapping register contents to characters and back. However, now that an orderly form of communication between programs and their runtime environment has been found, it can also be used to exchange more complex data objects. The idea is to use the stream of data structures from programs to the environment to describe requests for interactions and to use the reverse stream to describe responses of interactions. This allows to address different devices in the stream of requests or to indicate success or failure of interactions directly in the stream of responses. Since both input and output devices can be addressed in the request stream, synchronization of input and output operations can also be guaranteed, avoiding a whole class of nasty problems.

```
data Request = PutString String
             | GetString
             | ..

data Response = SUCCESS String
              | FAILURE String

main :: [Response] -> [Request]
```

Figure 3.6: Data types for request/response streams

Figure 3.6 presents the data structures used for requests and responses: the constructs of the request stream describe the interactions requested (kind of interaction, device, data), while the constructs of the response stream contain either the result of a successful interaction or a message describing the failure of an interaction.

The transformation system for request/response-streams is given in figure 3.7. The special variable `resp` represents the not yet available responses, but the most important difference in comparison to figure 3.3 is that input and output oper-

$$
\begin{array}{l}
\texttt{start}\ \lfloor \texttt{main} \rfloor\ ||\ < in, out > \\
\quad \mapsto\ (\texttt{main resp})\ ||\ < in, out > \\
C[\ expr\ ]\ ||\ < in, out > \\
\quad \mapsto\ C[\ expr`\ ]\ ||\ < in, out > \qquad\qquad\qquad\qquad , \ if\ expr \rightarrow expr` \\
\texttt{GetString} : expr\ ||\ < string, out > \\
\quad \mapsto\ [\ \texttt{resp} \leftarrow ((\texttt{SUCCESS}\ string) : \texttt{resp})\ ]\ expr\ ||\ < \Box, out > \\
\texttt{GetString} : expr\ ||\ < \texttt{eos}, out > \\
\quad \mapsto\ [\ \texttt{resp} \leftarrow ((\texttt{FAILURE "eos reached"}) : \texttt{resp})\ ]\ expr\ ||\ < \texttt{eos}, out > \\
(\texttt{PutString}\ string) : expr\ ||\ < in, \Box > \\
\quad \mapsto\ [\ \texttt{resp} \leftarrow ((\texttt{SUCCESS "PutString"}) : \texttt{resp})\ ]\ expr\ ||\ < in, string > \\
\texttt{[ ]}\ ||\ < in, \Box > \\
\quad \mapsto\ \texttt{done}\ ||\ < in, \texttt{eos} >
\end{array}
$$

Figure 3.7: Input/output with request/response streams

ations are no longer treated as different implicit transformations, but simply as services requested from different devices. Indeed, *all* interactions are explicitly requested through the program's result value and cause responses to be substituted for what was the program's parameter on startup (`resp`). While the list of responses is never terminated in this description, input operations may fail when no further input is available (fourth rule). All interactions are specified as synchronous here (causing immediate responses), but asynchronous interactions could be integrated into this system easily.

```
main resp =
    [ PutString "What is your name? ", GetString ] ++
    (hello (tl resp))

hello ( SUCCESS name : _ ) =
    [ PutString ("Hello, " ++ name ++ "!\n") ]
```

Figure 3.8: The login example with request/response streams

In the modified login function (figure 3.8), all interactions are explicitly requested in the result stream, and the success of the input operation is checked by pattern matching (it would be possible to add rules dealing with unsuccessful interactions). The program is defined as a list of the interactions involved, and no restructuring of the input is necessary to get at the first line of input text. Still, the unprocessed part of the responses needs to be passed on to `hello`.

The modifications needed for the dialog example (figure 3.9) are similar to those in the character stream version: `login` returns the unprocessed part of the response stream and the username as additional parts of its function value, and `main` does the necessary stream plumbing.

```
hello (SUCCESS name: rest) =
   (tl rest, [PutString ("Hello, " ++ name ++ "!\n")],name)

login resp =
   let
     req1 = [PutString "What is your name? ", GetString]
     (resp2,req2,name) = hello (tl resp)
   in (resp2,req1++req2,name)

dialog a b resp =
   (tl resp,[PutString ("player A: "++a++" player B: "++b++"\n")])

main resp = let
              (resp1,req1,a) = login resp
              (resp2,req2,b) = login resp1
              (resp3,req3) = dialog a b resp2
            in req1++req2++req3
```

Figure 3.9: The dialog example with request/response streams

Request/response streams are more flexible (in the kinds of interaction provided) than simple character streams, but share with them most of their other problems: a demand driven reduction strategy is still necessary, and though input and output are synchronized, in this variant it is the synchronization of requests and responses which is prone to errors. Every attempt to examine a response before the corresponding request has been generated would cause the program to deadlock. Furthermore, functions written in this style are still not composable without modifications, and programs tend to get cluttered with the details of stream handling, which restricts flexibility (in program construction).

When using this approach, it should be possible to write programs for many typical input/output-problems, but an additional level of abstraction is needed to hide the details of stream handling and to ease up the composition of programs involved in interactions. However, since the style is cumbersome to use without abstractions, it may be better to support the abstractions directly. Before we pursue this idea further, we investigate an alternative way of utilizing function parameters and function values to solve the input/output-problem.

3.3.2 Environment passing

So far, function values have been used either directly to hold the list of outputs or a list of interaction descriptions, and the parameters have either been the list of inputs or the list of interaction results. The second variant can be thought of as describing a communication between the program and an operating system, where only the latter has access to the system state. Alternatively, the program could access the system state directly or, to be precise, a representation of the system state. The basic idea is that functional programs describe transformations of the system state, albeit at a coarser, more abstract level than it is common in imperative languages. Functions get a representation of the current system state as parameter and return a representation of a new system state as their value (cf. figure 3.10).

$$
\begin{array}{l}
\texttt{start}\ \lfloor \texttt{main} \rfloor\ ||\ < in, out > \\
\qquad \mapsto\ (\texttt{main}\ (in, out))\ ||\ < \Box, \Box > \\
C[\ expr\ ]\ ||\ < \Box, \Box > \\
\qquad \mapsto\ C[\ expr`\ ]\ ||\ < \Box, \Box >,\ if\ expr \rightarrow expr` \\
(in, out)\ ||\ < \Box, \Box > \\
\qquad \mapsto\ \texttt{done}\ ||\ < in, out >
\end{array}
$$

Figure 3.10: State transformations with functional programs

However, this simple world-in/world-out model is not sufficient in many respects. The first problem is that the interactions are again restricted to program startup and termination. Non-strict evaluation would not help much in this case, unless inputs and outputs are represented as non-strict streams in the representation of the system state. What is needed instead is a way to let the functional program describe a state transformation and still be active (and thus able to generate further transformations of the state in response to changes made by other agents acting on the state, e.g., input devices) after the interaction.

One possible way to achieve this is to track the modifications of the system state representation during reduction and to reflect every change immediately in the state of the concrete system, effectively synchronizing transformations of the system state with program transformations that modify the state representation[1]. Whenever the program modifies the representation (by an output operation), the state of the concrete system has to be updated, too, and whenever the program executes an input operation, it becomes aware of any modifications of the system state (and its representation) caused by external agents (cf. figure 3.11). Once

[1]Barendregt [Bar96a] nicely characterized this as the umbilical cord between (formerly autistic) functional programs and the outside world.

again, input and output operations are just a special kind of state transformations.

$$
\begin{array}{l}
\texttt{start}\ \lfloor \texttt{main} \rfloor\ ||\ < in, out > \\
\qquad \mapsto\ (\texttt{main}\ (in, out))\ ||\ < in, out > \\
C[\ expr\ ]\ ||\ < in, out > \\
\qquad \mapsto\ C[\ expr`\ ]\ ||\ < in, out > \qquad\qquad , \ if\ expr \rightarrow expr` \\
UWC[\ (\texttt{getString}\ (string, out))\ ]\ ||\ < string, out > \\
\qquad \mapsto\ UWC[\ ((\texttt{SUCCESS}\ string), (\Box, out))\ ]\ ||\ < \Box, out > \\
UWC[\ (\texttt{getString}\ (\texttt{eos}, out))\ ]\ ||\ < \texttt{eos}, out > \\
\qquad \mapsto\ UWC[\ ((\texttt{FAILURE "eos reached"}), (\texttt{eos}, out))\ ]\ ||\ < \texttt{eos}, out > \\
UWC[\ (\texttt{putString}\ string\ (in, \Box))\ ]\ ||\ < in, \Box > \\
\qquad \mapsto\ UWC[\ ((\texttt{SUCCESS "putString"}), (in, string))\ ]\ ||\ < in, string > \\
(in, \Box))\ ||\ < in, \Box > \\
\qquad \mapsto\ \texttt{done}\ ||\ < in, \texttt{eos} >
\end{array}
$$

Figure 3.11: Input/output with environment passing

There is still one problem to solve in order to make this approach work: if a one-to-one correspondence between the representation and the real system state has to be established, it is essential that there is always *exactly one representation of the state of the outside world* when the functional program attempts an interaction. This is indicated in figure 3.11 by the use of *unique world contexts* (UWC). It is simply not possible (with current technology) to make any copies of the hardware at program runtime or to annihilate (a copy of) the hardware. There are several possible ways to solve this problem, but for now, we stick with our current problem, input/output, and assume that the proper use of an environment representation can be checked by a global analysis of the program text. The representation of the current state of the environment is only passed as a parameter to programs that have been checked this way. The types of the respective parameters are annotated as *unique* (`*`) and the type for environment representation is `*World`.

As shown in figure 3.12, primitive interactions are no longer described by data structures, but are represented as primitive functions. Just as the primitive pseudo-functions of our first approach, their evaluation may have side-effects on the global system state, but this cannot cause problems here because all side-effects are reflected by transformations of the local state representation. Furthermore, there is only a single representation of the state in the whole program, and every primitive interaction takes one state representation as a parameter and returns another one as part of its function value, so the order of interactions is fully determined by data dependencies. In contrast, other program transforma-

```
putString :: String -> *World -> *(Response, *World)
getString :: *World -> *(Response, *World)

data Response = SUCCESS String
              | FAILURE String

main :: *World -> *World
```

Figure 3.12: Data types for environment passing

tions are still only partially ordered, even though this flexibility may be restricted by the need to comply with the unique world requirement.

Using a uniqueness-checked environment passing style, the login procedure can be specified as a sequence of environment accesses, each one returning a modified environment and an additional value (the successful execution of each input/output-operation is again checked by pattern matching). The ordering of interactions is expressed through data dependencies only (regarding the `*World` values w_i), but the recursive definitions reflect the sequence of interactions (cf. figure 3.13).

```
main w0 =
 let
   (SUCCESS op1 , w1) = putString "What is your name? " w0
   (SUCCESS name, w2) = getString w1
   (SUCCESS op3 , w3) = putString ("Hello, " ++ name ++ "!\n") w2
 in w3
```

Figure 3.13: The login example with environment passing

Only a small modification is needed to reuse this function for the dialog example (the name has to be included in the function value), and the composition of programs involved in interactions is reasonably simple, too (cf. figure 3.14), if both primitive and complex interactions are defined to use the same interface. They take objects of type `*World` as parameters and return tuples consisting of results and possibly modified objects of type `*World` as their values.

The main disadvantage of this style are the ubiquitous values of type `*World`. The *explicit environment passing* that gives this style its name can be very tedious, especially under the restrictions of a *static uniqueness type checker*. Just like the request/response streams, uniqueness typed explicit environment passing seems

```
login w0 =
 let
   (SUCCESS op1 , w1) = putString "What is your name? " w0
   (SUCCESS name, w2) = getString w1
   (SUCCESS op3 , w3) = putString ("Hello, " ++ name ++ "!\n") w2
 in (name,w3)

dialog a b w =
   putString ("player A: "++a++" player B: "++b++"\n") w

main w0 = let
            (a,w1) = login w0
            (b,w2) = login w1
          in dialog a b w2
```

Figure 3.14: The dialog example with environment passing

to be expressive enough for many typical input/output-problems, but programs written in either of these styles are overloaded with unnecessary details. This should not pose great difficulties in a functional language: once the basic problem has been solved, it should always be possible to define suitable abstractions that hide the details of stream handling or environment passing. However, it is possible to devise a style of interaction that uses such abstract constructs as primitives.

3.4 Towards abstract descriptions of interactions

From hindsight, we have emphasized the problems of sequencing and composition of interactions in our examples and discussions, and we can try to address these two issues directly when defining abstractions for input/output. The idea is that programmers are only interested in primitive interactions and in means for the construction of complex interaction-based programs, whereas the details of how sequencing of interactions is achieved (via stream concatenation or via function composition) are better hidden in the definitions of more abstract operations. As mentioned before, such abstractions could also be defined on top of any of the previously discussed input/output-schemes to avoid (or hide) their problems.

The major problems with request/response streams are synchronization problems between requests and responses, clumsy stream handling and problems with composition. The common root of all these problems is the idea that there should be a list each for all requests and all responses, which are syntactically completely unrelated to each other. It has already been noted that this approach describes

a communication between the program and an operating system, and in this view, the possibly infinite streams simply represent communication buffers of unbounded capacities. Moreover, these buffers are fully visible in the program and have to be manipulated explicitly, resulting in a global view of communication that is not much easier to handle than the direct manipulation of global system states. This situation can be simplified greatly if the capacities of both communication buffers are restricted to one: both processes are then tightly coupled, though not completely synchronized, and programs only have to deal with one pair of request and response at a time, leading to a local view of communication.

The major problems with the environment passing scheme are caused by the ubiquitous environment representations and by the uniqueness constraints imposed on them. Both result from the decision to internalize a representation of the outside world into functional programs, and to let the interactions between the internal representation and the outside world happen behind the scenes. If, instead, we decide to model the context-sensitive embedding of functional programs into external runtime environments explicitly, programs have to pass control to this environment for each transformation of the system state. The environment performs the state transformations (including those caused by other agents) and passes control back to the functional programs afterwards.

3.4.1 Result continuations – sequencing individual interactions

Following either of the two derivations above, we arrive at an approach to input/output that focuses on individual interactions and their composition. The result of a program invocation is the description of only one interaction request and of the intended *continuation* of computation after the interaction (hence the name *result continuations*). The continuation expects the response to this particular interaction request as its only parameter and is thus a *continuation function*.

To summarize the modifications (cf. figure 3.15): the result of evaluating `main` is a data structure that describes the first interaction request and contains a continuation function which will be applied to the result of the first interaction. The body of the continuation is recursively constructed in the same way, possibly terminated by the empty continuation, `Done`. So, instead of one function dealing with many responses, as in the stream-based approaches, there are many functions dealing with one response each. Compared with environment passing, the program does not include and modify a description of the outside world, but explicitly describes its requests for interaction to an external environment in which it is embedded.

The transformation system (figure 3.16) is greatly simplified by this modified approach to input/output: no unique contexts or special variables are needed.

```
data Result = PutString String (Response -> Result)
            | GetString (Response -> Result)
            | ..
            | Done

data Response = SUCCESS String
              | FAILURE String

main :: Result
```

Figure 3.15: Data types for result continuations

Indeed, interactions with the system state occur only if the descriptions of these interactions (result continuations) are embedded in an empty program context, i.e., if they are textually located at the border between the functional program and its environment.

$$
\begin{array}{l}
\texttt{start}\ \lfloor \texttt{main} \rfloor\ ||\ < in, out > \\
\qquad \mapsto\ \texttt{main}\ ||\ < in, out > \\
C[\ expr\ ]\ ||\ < in, out > \\
\qquad \mapsto\ C[\ expr`\ ]\ ||\ < in, out > \qquad , \ if\ expr \rightarrow expr` \\
(\texttt{GetString}\ cont)\ ||\ < string, out > \\
\qquad \mapsto\ (cont\ (\texttt{SUCCESS}\ string))\ ||\ < \Box, out > \\
(\texttt{GetString}\ cont)\ ||\ < \texttt{eos}, out > \\
\qquad \mapsto\ (cont\ (\texttt{FAILURE "eos reached"}))\ ||\ < \texttt{eos}, out > \\
(\texttt{PutString}\ string\ cont)\ ||\ < in, \Box > \\
\qquad \mapsto\ (cont\ (\texttt{SUCCESS "PutString"}))\ ||\ < in, string > \\
\texttt{Done}\ ||\ < in, \Box > \\
\qquad \mapsto\ \texttt{done}\ ||\ < in, \texttt{eos} >
\end{array}
$$

Figure 3.16: Input/output with result continuations

The specification of the login procedure (figure 3.17) becomes even simpler in this style, and the continuation functions seem to be particularly well suited to describe the 'what to do next?'-idea of interaction sequencing (wildcard patterns '_' are used to express that results of `PutString`-interactions are not inspected).

Nevertheless, the function has to be rewritten before it can be reused for the dialog example (figure 3.18). Instead of being terminated with `Done`, it takes an extra continuation parameter (`cont`) that is applied to the result of the `login-`

```
main = PutString "What is your name? " \_->
       GetString \SUCCESS name->
       PutString ("Hello, " ++ name ++ "!\n") \_->
       Done
```

Figure 3.17: The login example with result continuations

interaction (the username).

```
login cont = PutString "What is your name? " \_->
             GetString \SUCCESS name->
             PutString ("Hello, " ++ name ++ "!\n") \_->
             cont name

dialog a b cont =
    PutString ("player A: "++a++" player B: "++b++"\n") \_->
    cont

main = login \a->
       login \b->
       dialog a b  \_->
       Done
```

Figure 3.18: The dialog example with result continuations

Whereas the composition of primitive interactions is built into the syntax of result continuations, complex programs involved in interactions need to be rewritten into a *continuation passing* style before they can be composed. Compared with the explicit environment passing style, the gain seems to be small, as environment passing has been replaced by continuation passing, but at least these continuations do not have to be unique. Still, the need to pass them around explicitly is annoying, and is usually avoided by defining a general composition operator for interactions.

A closer look at the primitive constructs of this style reveals the causes of this problem. The primitive constructs are not elementary: each primitive result continuation describes both an elementary interaction and a continuation function. Using only the basic constructs, it is simply not possible to specify an interaction without explicitly mentioning what should happen after it. Since this information is usually not available where the primitive interactions are used,

but belongs to other program parts, the continuations need to be passed as parameters when program parts are composed together. This leads to continuation parameters scattered all over the program text. The continuation part is similar for all primitive constructs, but is repeated in each of them, and composition should work equally well for primitive interactions and for complex interaction sequences, but it does not.

3.4.2 Monadic style – composing scripts of interactions

As a first step towards better support for program composition, the basic constructs of the result continuation style can be broken up into their elementary parts, the first one a description of one particular primitive interaction and the second one a composition with a continuation function. The first part can be derived from the corresponding result continuation construct by simply omitting the continuation parameter, while the second part is the same for all interactions, so that one additional construct suffices to describe the same information. The new construct is called `Bind`, as it is used to bind together an interaction description with a continuation function (cf. figure 3.19).

```
data IO = PutString String
        | GetString
        ..
        | Bind IO (Response -> IO)
        | Unit Response

data Response = SUCCESS String
              | FAILURE String

main :: IO
```

Figure 3.19: Data constructors for monadic style

Of course, the evaluation of the two parts has to be separated, too, for otherwise the change in syntax would not make any difference. The second new construct, `Unit`, is needed to describe the intermediate expression between the two steps: the interaction has happened and has left a return value, which is the parameter of `Unit`. `(Unit x)` is itself an interaction description and may thus be combined with a continuation using `Bind`, but it describes an identity interaction, i.e., performing `Unit x` does nothing but returning `x`. With these prerequisites, it is possible to describe the two evaluation steps: first, the interaction description, which is the first parameter of a `Bind`-construct, is performed and the return

value is embedded into a `Unit`-construct. The `Bind`-construct itself is only transformed if its first parameter is such a `Unit`-construct, from which the return value is taken and passed as a parameter to the continuation function.

So far, it may seem as if this was only a reformulation of the result continuation style in more elementary steps. However, having isolated the elementary constructs and evaluation steps, it becomes possible to combine them in new ways, thus creating a more flexible scheme of programming with interactions. The important insight here is to realize that every complex interaction terminates with a primitive one and will therefore eventually be transformed into a `Unit`-construct. Consequently, there is no reason to restrict the first parameter of `Bind` to primitive interaction descriptions, and `Bind` can be used to compose both primitive and complex interaction descriptions with continuation functions.

Formally, this can be specified by giving the same status to the `Bind`-construct as to the primitive interactions (cf. figure 3.19). Informally, programs still return data structures describing the interactions to be performed, but these structures are shaped like binary trees instead of binary lists (which was the case not only with request/response-streams but also with result continuations), and this greatly simplifies the composition of complex interaction descriptions. The trees are interpreted in depth-first traversals, performing each primitive interaction in sequence. The construction of interaction descriptions by program transformations and their evaluation by interactions with the global system state are interleaved, just as in the result continuation approach.

```
main = Bind (PutString "What is your name? ") \_->
       (Bind GetString \SUCCESS name->
        PutString ("Hello, " ++ name ++ "!\n") )
```

Figure 3.20: The login example with monadic data constructors

A first variant of the login specification (figure 3.20) looks very similar to the result continuation version, with additional `Bind`s to compose descriptions of primitive interactions with continuation functions and no terminating `Done`. The latter is important, because it means that no pre-emptive assumptions are made about whether or not the interaction description will be composed with a continuation.

Before we give the formal transformation system for this input/output-style, we use the opportunity to make some convenient syntactic changes. In particular, we introduce a symbolic infix operator (`>>=`) for `Bind` and rename `Unit` to `return`. All interactions are given as constructors of an abstract data type `IO`. We also introduce a variant of `>>=` that ignores the intermediate return value of its first parameter (`>>`). The modifications are summarized in figure 3.21.

```
putString :: String -> IO
getString :: IO
main :: IO

infix >>,>>=
>>= :: IO -> (Response -> IO) -> IO
return :: Response -> IO

>> :: IO -> IO -> IO
a >> b = a >>= \_-> b
```

Figure 3.21: Data types for monadic style

The advantage over the algebraic data type used before is that we have better control over the means for inspection of interaction description (no pattern matching over the internal structure of `IO`). We therefore abstract from the concrete representations of interaction descriptions in functional programs.

$$
\begin{array}{l}
\texttt{start}\ \lfloor \texttt{main} \rfloor \ ||\ < in, out > \\
\qquad \mapsto\ \texttt{main} \ ||\ < in, out > \\
C[\ expr\ ] \ ||\ < in, out > \\
\qquad \mapsto\ C[\ expr`\ ] \ ||\ < in, out > \qquad\qquad , \ if\ expr \rightarrow expr` \\
MC[\ \texttt{getString}\ ] \ ||\ < string, out > \\
\qquad \mapsto\ MC[\ (\texttt{return}\ (\texttt{SUCCESS}\ string))\ ] \ ||\ < \Box, out > \\
MC[\ \texttt{getString}\ ] \ ||\ < \texttt{eos}, out > \\
\qquad \mapsto\ MC[\ (\texttt{return}\ (\texttt{FAILURE "eos reached"}))\ ] \ ||\ < \texttt{eos}, out > \\
MC[\ (\texttt{putString}\ string)\ ] \ ||\ < in, \Box > \\
\qquad \mapsto\ MC[\ (\texttt{return}\ (\texttt{SUCCESS "putString"}))\ ] \ ||\ < in, string > \\
MC[\ ((\texttt{return}\ result)\ \texttt{>>=}\ cont)\ ] \ ||\ < in, out > \\
\qquad \mapsto\ MC[\ (cont\ result)\ ] \ ||\ < in, out > \\
(\texttt{return}\ expr) \ ||\ < in, out > \\
\qquad \mapsto\ \texttt{done} \ ||\ < in, \texttt{eos} >
\end{array}
$$

Figure 3.22: Input/output in monadic style

With these modifications, the transformation system is defined in figure 3.22. Apart from the new monadic contexts MC, the rules for the primitive interactions are simplified, and the common behavior of passing on an intermediate result to a continuation function is defined in a separate rule for `>>=` (sixth rule).

The use of monadic contexts accounts for the new composition structure of interaction descriptions: whereas the first interaction is described in the outermost constructor of a result continuation, it is described in the first leaf of a binary tree with monadic style constructs. This is formalized in the definition of monadic contexts for interactions:

$$MC \;=\; [\,] \mid (MC \;>\!>\!=\; expr)$$

A specification of the login procedure using the modified syntax is presented in figure 3.23. Note how the program is reduced to the essential tasks of describing the primitive interactions and their composition, and how the composition of interaction descriptions also specifies their relative ordering.

```
main = putString "What is your name? " >>
      (getString >>= \SUCCESS name->
       putString ("Hello, " ++ name ++ "!\n"))
```

Figure 3.23: The login example with monadic style input/output

This interaction description can be used as part of more complex descriptions without modifications, and only an identity interaction carrying the username needs to be added to make the name available as an intermediate result. Composition of both primitive and complex interaction descriptions is simply done by `>>` or `>>=`, and no additional syntactic baggage for environment, continuation or stream handling is needed to specify the dialog program (cf. figure 3.24).

```
login = putString "What is your name? " >>
        getString >>= \SUCCESS name->
        putString ("Hello, " ++ name ++ "!\n") >>
        return name

dialog a b =
    putString ("player A: "++a++" player B: "++b++"\n")

main = login >>= \a->
       login >>= \b->
       dialog a b
```

Figure 3.24: The dialog example with monadic data constructors

Compared with request/response streams, result continuations, and environment passing, monadic input/output seems to be a more declarative way to specify interactions. Nothing needs to be said about how other language constructs (such as functions and their composition or lists and their concatenation), language properties (strict versus non-strict), or features of type systems (uniqueness typing) have to be used to achieve sequencing and composition of interactions. Nor are programs burdened with low-level details (e.g., of stream handling) in order to define what should be primitive interactions. Programs using monadic style just describe which interactions are to be performed in which order.

The notation for interactive programs written in the monadic style is irritatingly close to the notation used in imperative languages. This should not be too surprising, however, since the imperative notation was designed for the purpose of interaction (with the state of a machine, initially; with the state of a store of box variables, later). Note also that the monadic notation only *includes* the usual imperative notation for the most elementary programs. Beyond this, it allows various possibilities for program construction that are not available with usual imperative notation. For instance, descriptions of interactions are first-class data objects, can be stored in data structures, or dynamically composed into larger interaction scripts. Moreover, functional abstraction can be used to define control abstractions that, in conventional imperative languages, need to be built into the language definition. Functions that return interaction scripts correspond to procedures, conditional, interaction-valued expressions correspond to conditional statements, and different variants of interaction loops can easily be defined via recursive functions. Of course, functions can also take interaction scripts as parameters.

3.5 The concept of monads

We have introduced the constructs of the monadic programming style as a declarative way of describing programs involved in interactions, but have not yet described the concept of monads. The concept originated in category theory, and was employed by Moggi [Mog89b, Mog89a] to give a categorical semantics for computational aspects of the λ-calculus, especially if extended with additional features and used as a tool in the study of programming languages. While Moggi strived for an abstract view of programming languages on the semantic level, Wadler [Wad92a, Wad92b] translated the ideas to the language level, which finally turned monads into a widely used technique for structuring functional programs (cf. also [HC94]). Following [Wad92b], a monad in a functional language is represented by a triple `(M, unitM, bindM)` consisting of a type constructor `M` and a pair of polymorphic functions

```
unitM :: a -> M a
bindM :: M a -> (a -> M b) -> M b
```

`unitM` has to be a left and right unit for `bindM`, and `bindM` has to be associative in order for such a structure to qualify as a monad (*monad laws*). The basic idea of using monads, shared by semantic accounts and functional programming techniques, is to distinguish between simple values and computations that return values (but may have some additional computational effect). In functional language terminology, this distinction is based on types, and the type constructor `M` serves to describe the kind of computation. `unitM` allows to embed simple expressions into identity computations (returning the expressions without having effects in the kind of computation considered), and `bindM` allows to compose a computation and a continuation function to form a new computation.

In our case, the computations we are interested in are interactions with some external environment that return expressions, and the monad in question consists of a type constructor that forms the type of interaction scripts from the type of expressions and the interaction constructors `return` and `>>=`. Whether or not these two constructors obey the monad laws mentioned above depends in part on the way in which the functional sub-system (defined by the transformation rule given in figure 3.22) is embedded into a larger system comprising other agents (including, e.g., communication devices and other functional processes). If only the rules of the functional sub-system are considered, the rule for `>>=` directly defines `return` to be a left unit for `>>=`, and it follows from the same rule that `return` is also a right unit. For associativity (of `>>=`) to hold, it is important that the structure of an interaction script has no effect on the computation, so that only the sequence of primitive interactions is important. This is true for the functional sub-system: firstly, by the use of monadic interaction contexts, the first primitive interaction in a script is always executed first (regardless of and without changing the exact structure of interaction scripts). And secondly, the only rule for `>>=` (which eliminates a local part of the script structure to make the next primitive interaction the first one) neither depends on nor changes the state of the external input/output-registers.

What has made the concept of monads so successful, both in language semantics and as a basis for functional programming techniques, is its generality: simple structures, each consisting of a type constructor and two operations, bind and unit, are suitable to model a large variety of language features, including input/output, state transformations, exception handling, and non-deterministic computations (cf. Wadler's papers for examples of functional programming techniques involving these features, and Moggi's papers for examples of semantic accounts of programming languages having these features). So these constructs do not only abstract from the exact implementation of computations (as in our presentation of interactions) but also from the exact nature of effects (input/output, state transformations, exceptions, ...). While there are other ways to address each particular kind of computation mentioned above, monads provide a uniform, abstract framework for the whole class, and the monad laws facilitate reasoning about programs written in monadic style.

3.6 Summary and related work

Starting from a very sketchy problem specification, we have been able to develop basic variants of all major approaches to input/output in purely functional languages. The steps from one style to the next are reasonably small and can be derived from a careful analysis of the problems encountered. Though the historical development may not have proceeded this way, the presentation shows that language design does not necessarily need a flash of genius to solve problems as new and fundamental as the one in question here. As there are still other problems to be solved in the design of programming languages, we find this very encouraging.

The recurring theme of the presentation of input/output-schemes for functional languages in this chapter is the integration of context-sensitive transformation rules (that describe interactions of programs with an external environment) with the context-free rules that describe the evaluation of functional programs. A simple combination of both kinds of rules, by the use of pseudo-functions, has proven to be inadequate as it invalidates characteristic properties of functional languages and requires a total ordering on both kinds of rules in order to guarantee deterministic sequences of interactions. Consequently, all other approaches limit the contexts in which interactions may occur, and connect the ordering of interactions to the values returned by functional programs. Character streams fail to define the ordering of input interactions relative to output interactions, and request/response-stream, while correcting this shortcoming, do not guarantee that there is a request for each response that is demanded during program evaluation. Both stream-based approaches suffer from taking a global view of communication (programs have access to lists of all interactions), while the remaining approaches focus on individual interactions, thereby facilitating the composition of programs involved in interactions. Environment passing is a very flexible scheme for input/output, because there are only few restrictions for the contexts in which interactions may occur. However, interactions are not explicitly distinguished from other program transformations, so that the context restrictions extend to all kinds of transformations. Result continuations, on the other hand, do make such a distinction, but are extremely restrictive about the permissible interaction contexts. These have to be empty, causing a complete program continuation to be packed into the description of each primitive interaction (contrast this with environment passing, where descriptions of primitive interactions can appear almost anywhere in program continuations). Finally, monadic style also separates interactions from other program transformations, but considerably relaxes the restrictions on interaction contexts.

Just as a mathematical proof tries to make evident the truth of a theorem by providing a derivation of the theorem from basic axioms by comprehensible steps, our presentation of input/output-schemes as steps in a line of development tries to make evident why current schemes of input/output are as they are, and which

design decisions have lead to their current form. Not only does this help to understand these schemes and their relationships, it also gives confidence that they really fulfill the problem specification, and it shows which design questions would have to be answered differently if one would set out to develop another style of input/output. In the remainder of this section, we try to complete this conceptual overview of the problems and solutions regarding functional input/output with a brief account of the history of input/output in functional languages, and provide some references to the actual work that has been done in this area. Note that, though we hope to provide a rough overview of the history and diversity of the topic, this section is not intended to include a complete list of references. Further reference to historical material can be found in [HS89, JS89, Per91, Gor94, Sch93].

Since the early 1960s, side-effecting input/output-primitives have been used as pseudo-functions in Lisp [MAE+66, Appendix F], a language with a strong functional flavor but imperative in nature[2]. Today, the ANSI Common Lisp Standard [X3J94] still defines a sequential order of evaluation both for explicit control structures and for the arguments of function calls. In contrast, the order of evaluation is explicitly left unspecified for the arguments of function calls in the Lisp dialect Scheme [Cli91] – it is only constrained to be consistent with some sequential order of evaluation. While this underspecification does not facilitate reasoning, it discourages programming styles that do rely on specific evaluation orders, and liberates implementations from the need to guarantee a specific sequencing of side-effects for programs that depend on evaluation order (beyond the call-by-value regime which is required by the Scheme report). Standard ML [MTH90, MT91] is another modern functional language that uses pseudo-functions for input/output and its definition has to be explicit about the total order of evaluation for each and every syntactic construct (including records and collections of declarations) in order to guarantee a deterministic semantics. Note that some of these languages refer to streams to describe the sequences of input and output items, but use side-effecting pseudo-functions to get items from input streams or to put items into output streams. All these languages are not referentially transparent.

Streams as a basis for the formal definition of input/output were proposed as early as 1965 by Landin [Lan65], who used λ-abstractions as non-strict contexts for unevaluated parts of streams (his correspondence between Algol 60 and the λ-calculus was based on an SECD-machine extended with imperative features; thus the order of evaluation was defined through a call-by-value regime towards weak head normal forms). However, he noted the problems of abstracting over expressions that may not have weak head normal forms under a call-by-value regime. It took until the late 1970s before Henderson and Morris [HJHM76] presented

[2]Contrary to common belief, Lisp was never intended as an implementation of the λ-calculus (cf. [McC81]), and despite McCarthy's motive to 'allow proofs of properties of programs using ordinary mathematical methods' and the existence of pure Lisps, the main line of the Lisp family of languages relied on side-effects for efficiency reasons.

a lazy evaluator that made Wadsworth's call-by-need evaluation regime [Wad71] available for a purely functional subset of Lisp, providing better means to handle potentially infinite streams (cf. also [FW76]). Soon afterwards, Henderson [Hen80] explored the potential of streams and lazy evaluation and described networks of functional processes communicating via potentially infinite streams. In the following years, attempts to give (mostly) functional specifications of operating systems and networks of functional processes became the driving forces that lead to the specification and implementation of input/output-systems based on streams for purely functional languages (for further references, cf. section 4 of [Sch93], and also [JS89]).

Just as stream-based approaches, continuations appeared first in formal descriptions of programming languages. In the early 1970s, Strachey and Wadsworth used continuations to describe imperative language features in denotational semantics [SW74]. While Wadsworth coined the name 'continuation', the concept of continuation functions was discovered several times under different names (cf. [Rey93] for an account of this phenomenon). The translation of continuation functions from semantics to functional languages was simple[3], and they were soon put into use to simulate imperative features in Scheme by Steele and Sussman [GLSS76] (who also note that continuation functions have already been used by Church [Chu51] to model data structure selectors in the λ-calculus). However, continuations were not readily accepted as a means to model input/output in purely functional languages.

In the late 1980s, programming practice indicated that the seemingly elegant stream based approaches were rather ill suited for the specification of complex systems, and abstract combinators were developed to hide the details of stream plumbing, thus avoiding some of the possible problems in program development. For instance, Thompson [Tho90] presented such a set of combinators as 'the basis for a disciplined approach to writing *[interactive programs]*'. His combinators, defined on top of a stream-based input/output-scheme, had some similarities to the constructs of a monadic style of input/output– there was a dedicated type of interactions, a sequential composition operator for interactions, and functions to read and print characters. The crucial point here is that the second parameter of the sequential composition operator was a continuation function. According to [Gor94, chapter 1.2] and [HS89, section 1.1], several other continuation-based approaches were in use during the 1980s. It was Perry [Per91], who coined the term 'result continuation' and implemented them as the basic input/output-mechanism in Hope$^+$C. Nevertheless, the apparent simplicity of stream-based approaches and their close relation to lazy evaluation prohibited a general switch from stream- to continuation-based input/output-schemes. For instance, Jones and Sinclair [JS89] note that 'Result continuations are very similar, and possibly identical to

[3]although efficient implementations were needed for the *tail calls* that occur frequently in a continuation passing style of programs (cf. [GLSS76])

what can be provided *by a suitable packaging* of streams.' (emphasis added), but continued: 'Overall, we judge that streams provide the best workable basis for an I/O architecture at present'. Similarly, Hudak and Sundaresh [HS89], who compared the expressiveness of both styles and gave translations between them, note that 'the continuation style is often easier to use and the resulting programs easier to read', but then go on to specify the input/output-system of Haskell, which was based on request-/response-streams. They comment: 'This does not mean that streams are the preferred programming model, but just that they are considered simple and general enough to be designated as primitive'. A continuation-based input/output-system was provided as a layer of abstraction on top of request/response-streams.

Finally, monads were proposed to address several related problems in a uniform way (cf. section 3.5). This time, the transfer of tools for the definition of language semantics to the domain of functional programming proceeded rather rapidly: Moggi's categorical semantics of computational λ-calculi [Mog89b, Mog89a] appeared in 1989, Walder's translation of the concept into a functional programming technique first appeared in 1990, and was refined [Wad92a] and simplified [Wad92b] in 1992. In 1993, Peyton Jones and Wadler [PJW93] presented 'a new model, based on monads, for performing input/output in a non-strict, purely functional language. It is composable, extensible, efficient, requires no extensions to the type system, and extends smoothly to incorporate mixed-language working and in-place array updates.'. Since they also outlined an efficient implementation scheme, *imperative functional programming* (the title of the paper) based on monads was rapidly accepted as 'the' new approach to input/output in purely functional languages. After a period of unofficial extensions to Haskell implementations, the Haskell specification switched to monadic input/output as the basic scheme with version 1.3 of the Haskell report [PH96] in 1996.

Independently, the Clean [PvE97] group adopted an environment passing style for input/output. This was made feasible by Clean's static uniqueness type inference system [BS95], which allows to check the single-threaded use of environments statically. The approach is described in Achten's PhD thesis [Ach96], where several extensions and applications are also presented. Most notably, an event-driven, graphical I/O-system is developed, which allows a declarative approach to the specification for graphical user interfaces in Clean. The unique environment is hierarchically decomposed into unique sub-environments (e.g., for the file system and the devices of the graphical user interface), and interactions with each of these sub-environments can proceed independently of each other (the type system ensures that programs re-compose the sub-environments to a complete environment before they terminate).

Chapter 4

Module Systems for Functional Languages

Modular programming can be characterized as a style of programming that organizes large programs into smaller structures in such a way that they are easier to understand, to maintain, and to reuse. There is a simple intuitive description of modules, with which most programming language designers and programmers will agree, namely that

modules are building blocks for complex programs,

but everybody seems to have his or her own interpretation of what a building block should be and what kind of linguistic support for modular programming should be provided. Furthermore, as outlined in the introduction, the basic understanding of what a program is has developed over time and will continue to do so in the future. Of course, this affects our ideas of what program building blocks are and how they may be composed to form complex programs. Last, but not least, the development of higher level programming languages allows to specify solutions to what would have been called complex problems before in terms of rather simple programs, thus providing the same kind of complexity reduction that modules should offer.

However, to quote from [Dij72], 'The increased power ... made solutions feasible that the programmer had not dared to dream about a few years ago. And now, a few years later, he had to dream about them and, even worse, he had to transform such dreams into reality!'. Dijkstra talked about the increased power of the hardware and tried to identify the origins of the so called software crisis. Nevertheless, the same argumentation applies just as well to our situation today, and the same issues of program complexity that led to the development of linguistic support for modular programming around 1970 force us to reinvestigate these topics today. The complexity of the problems we attempt to solve using modern programming languages has increased up to a level that no longer seems

to be adequately compensated by the higher level of abstraction supported by these languages. Indeed, an end of the software crisis has never been declared.

In contrast to the situation for input/output, we cannot treat language support for modular programming as an isolated problem in the domain of functional languages. We cannot even provide some out-of-the-box requirement specification. Any idea of what should make up a program building block reflects the current state of the art with respect to programs, programming languages and modular program design. Therefore, a historical approach to the topic seems advisable.

4.1 Some highlights in the history of modular programming before 1980

It is difficult to pin down the origin of modules, especially in view of the various possible definitions. Modular programming seems to have been around for a long time without receiving particular attention or even specific language support. Many experienced programmers knew that it was useful to decompose a problem into simpler ones during the program design phase. However, the conceptual gap between program design and the actual coding on some computer equipment was so wide that they did not even think of getting support for such techniques on the programming language level. On the other hand, the design phase was not formalized in a way that allowed such techniques to be disseminated widely. Consequently, the unfulfilled – for a long time even unexpressed – need for better language support was always ahead of the actual language development and every new language feature that could be used in any way to alleviate this mismatch was soon put into (mis-)use to implement the designs.

In the following, we try to illustrate the development of some of the most important ideas as they present themselves in a few influential publications. This is by no means an attempt to provide a complete overview of the history and the years of publication do not necessarily reflect the actual development in time. The main purpose is to introduce the various ideas of program modularity in their historical context, and thus the presentation will be very specialized here. Wegner's description of the first 25 years of programming language development [Weg76] may serve to provide a more general context. The presentation given here will already show some relations between the otherwise seemingly unrelated approaches to program modularity and will allow us to extract what we think is the essence of most of them later.

In early, low-level languages, programs were just sequences of machine instruction and program parts corresponded to subsequences. Extracting common subsequences from programs and specifying a standard way to jump into and out of such *closed subroutines* provided the earliest framework for modular pro-

gramming with language support. Closed subroutines, *subroutine libraries* and *assembly subroutines* that allowed 'the assembly of the master routine and the subroutines to be performed automatically by the machine' were described already in [WWG57].

Algol 60 [eBB+97] introduced the *block* as a structuring tool. A block consisted of a sequence of procedure and data declarations followed by a sequence of executable statements enclosed in begin-end parentheses. Blocks and procedures (which were just named parameterized blocks) could be arbitrarily nested, declarations local to a block were visible only inside that block and its inner blocks (provided it was not shadowed by an inner declaration for the same name).

The authors of Simula 67 [DMN70] recognized the block concept as 'the fundamental mechanism for decomposition in Algol 60', corresponding to 'the intuitive notion of "sub-algorithm" '. They extended the block concept to that of a *class* by allowing block instances to survive their call. In brief, classes were textually similar to procedures and they could be called with parameters which replaced formal parameters in the class definition to form a block instance which was then executed. However, on return from a class, the instance was not destroyed. Instead, a reference to it was returned to the caller: a way to access the variables, procedures and classes local to this particular block instance, called an *object*. Class declarations and simple blocks could be prefixed with other class identifiers, causing the defining blocks to be concatenated (classes defined this way were called *subclasses* of their prefixes).

Using the class mechanism, it was possible to model simple concepts, e.g., to specify data structures and to define operations on them. Program concatenation and the subclass mechanism then allowed to establish hierarchies of concepts and to compose programs from constituents found in such a hierarchy. These features were used heavily to adapt the common base language to special application domains (e.g., discrete event simulation) by defining suitable classes containing the necessary problem-oriented concepts. To prepare for separate compilation of procedures and classes, such items could be declared as external (recommended optional part of the common base language).

The description given above is based on a revised report on Simula 67 published in 1970. Earlier versions of the language [DN66] were more oriented towards the original application area, discrete event simulation, providing the concepts of an 'activity' (declaration for a class of processes) and a 'process' (dynamic instance of an activity declaration). In the revised version, these concepts were replaced by the more abstract 'classes' and 'objects' and all simulation-specific language constructs were replaced by an application-specific standard class 'simulation'. The authors explicitly acknowledged the influence of Hoare's ideas on records, record classes and references [WH66, Hoa68] on the development of Simula 67 (cf. also [DDH72]).

In his 1972 ACM Turing Award lecture [Dij72], Dijkstra directly addressed the intellectual manageability of programs and identified it as an independent

property, not being related to the mathematical content of programs, but being at the heart of the 'software crisis'. He mentioned closed subroutines, abstraction, hierarchical and nicely factored solutions as the key concepts in programming that could make it technically feasible to '... design and implement the kind of systems that are now straining our programming ability at the expense of only a few percent in man-years of what they cost us now ...', i.e., to design large sophisticated systems while keeping the programs intellectually manageable.

Dijkstra also emphasized the importance of notational support: 'I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language – our basic tool, mind you! – already escapes our intellectual control', 'I see a great future for very systematic and very modest programming languages', '... to a much greater extent than hitherto they *[tomorrow's programming languages]* should invite us to reflect in the structure of what we write down all abstractions needed to cope conceptually with the complexity of what we are designing.'.

Also in 1972, Parnas [Par72] wrote about 'the criteria to be used in decomposing systems into modules', discussing 'modularization as a mechanism for improving the flexibility and comprehensibility of a system, while allowing the shortening of its development time'. He identified decomposition into major processing steps (which he called flowchart abstraction) as the most common approach to modularization. He claimed that 'The flowchart was a useful abstraction for systems with on the order of 5,000-10,000 instructions, but as we move beyond that it does not appear to be sufficient; something additional is needed.'.

He proposed *information hiding* as an alternative criterion for decomposition in the sense that 'every module is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.'. This should lead to a decomposition into largely independent modules, restricting the effects of system changes and of revisions of design decisions to only a few modules, thus enhancing flexibility. Parnas lists independent development and comprehensibility of modules as two other expected benefits of modular programming that could be achieved by following his decomposition criterion and concludes 'that hierarchical structure and "clean" decomposition are two desirable but independent properties of system structure'.

Among the specific examples of decompositions which seemed advisable, Parnas mentioned: 'A data structure, its internal linkings, accessing procedures and modifying procedures are part of a single module.'. Identifying this kind of modules as important and developing linguistic support for them seems to be a main line of development towards modular programming at that time. Zilles [Zil73] used the term *procedural encapsulation* to describe a similar program structuring technique used in the development of the experimental operating system OS6 [SS72a, SS72b] and noted that 'Using the technique of encapsulation, a data

type is completely characterized by the set of operations defined on the data and the observable relationships between those operations'. Liskov and Zilles [LZ74] coined the term *abstract data type* and introduced special modules, called 'operation clusters', to support programming with these types in the language CLU [Lis92]. A cluster had three parts: a storage representation for objects of the new type in terms of existing types, a collection of procedures and a header defining which of these procedures could be called from outside of the cluster.

Many programming languages designed in the 1970s provided explicit support for modular programming, being influenced by Simula's classes or by the ideas of information hiding and abstract data types. Examples include Alphard [Sha81], Mesa [GMS77] and Modula [Wir77]. Only the module facility developed for the Modula family of languages will be described in some detail here as it seems to be typical for many of the module systems currently used for functional languages. Wirth [Wir79] described his modules as a simplified language construct for information hiding which 'has one and only one function, namely to establish a static scope of identifiers, whose acrossboundary visibility of identifiers is strictly under the programmer's control'. The concept was based on three main elements: 'The module is effectively a bracket around a group of ... declarations ... the import list contains those identifiers defined outside which are to be visible inside too, and the export list contains the identifiers defined inside that are to be visible outside.'. Later, qualified export (prefixing exported identifiers with the module name to avoid name clashings between multiple exports) and the distinction between *definition modules* and *implementation modules* were added. A definition module 'specifies the interface of a module, in particular all objects that are exported', whereas the implementation module 'belongs to the definition module carrying the same name. It contains the bodies of the procedures whose headings are listed in the definition module, and possibly declarations of further objects that are local, i.e. not exported'.

In 1976, DeRemer and Kron [DK76] explicitly distinguished the activities of writing large programs from that of writing small ones. They coined the term *programming-in-the-large* and advocated the introduction of separate *module interconnection languages* for this kind of task. Moreover, they argued 'that structuring a large collection of modules to form a "system" is an essentially distinct and different intellectual activity from that of constructing the individual modules' and that 'essentially distinct and different languages should be used for the two activities'.

We leave the early history of modules here, noting that we have deliberately left out some facets of modularity such as the partitioning of nonsequential systems, the interaction between concurrent programming and modular programming in the development of some language facilities, the field of software technology, the algebraic specification of interfaces, or the further development of object-oriented programming. Research and practice in these fields have contributed to the current view of modular programming, some of the fields even

developed into independent areas of computer science, but we want to concentrate on the basic structure of the problem. There will be more to say on the development of module facilities from 1980 to today later, but it turns out that many current functional languages, as far as their module systems are concerned, rely on the state of the art of 1980.

4.2 Conventional module systems for functional languages?

There seems to be nothing special about module systems for conventional programming languages. They allow large programs (viewed essentially as collections of declarations) to be organized into smaller, more comprehensible modules. The separation between these program parts is established via explicit control over the visibility of identifiers defined inside the parts. This view of module systems can of course be adopted for functional languages and this has been done, e.g., for the languages Clean [PvE97, version 1.2] and Haskell [PH96, version 1.3]. Both define programs to be collections of modules which are themselves collections of definitions, declarations and explicit references to other modules via `import` declarations. In Clean, interfaces and implementations have to be provided in separate files, control over exports is by repetition of (parts of) definitions in the interface files, cyclic dependencies of interface files are prohibited. In Haskell, export is controlled by explicit `export` declarations, separate interface files have been removed from the language definition, mutually recursive modules are allowed. Both languages allow to explicitly import an (exported) definition from another module by referring to the defined name or to implicitly import all exported definitions by referring to the module's name; Haskell also provides selective and qualified import and the use of local aliases for imported modules as means to resolve name clashes.

This is basically the Modula view of modules as additional structures (brackets) around collections of declarations with explicit control over the scope of the declared identifiers. Modules are viewed as static objects, i.e., they are not relevant to the dynamic semantics of the programming language and there are no constructs in the language that could deal with them as data objects directly (without resorting to source code manipulation). Therefore, a separate module language is needed in order to describe these structures and their interrelationships and to construct complex programs from these building blocks. This module language has two parts, the static one explicitly visible in the language definition, the dynamic one rather implicit and not part of the language definition: the static dependencies (import, export) between modules are described in the source text using extensions to the language syntax, whereas languages and tools of the programming environment are used to handle the dynamic aspects of mod-

ules (such as reorganization, version control, compilation and linking). The tools operate either on the source text or on a low level object code representation. They are not part of the programming language definition but reside at the operating system level. Even the parts of the module language that are interspersed among the constructs of the programming language's syntax really belong to a different domain, to a separate language level. It is mainly for the simplicity of the visible constructs that this separate level is seldomly recognized as forming (part of) a language in its own right, and it is partly because of this fact that module languages are not often extended with more sophisticated constructs for program construction.

Now what is wrong with this? Early high-level languages were merely thought of as input data formats for compilers producing machine code modules, with the machine level and later the operating system level being the only real levels for programmers to work on. In other words, while the task of programming itself was relieved from low-level details, any tasks in which it was necessary to work with programs as data (compilation, linkage, execution, management of program and module libraries, etc.) were not. In particular, this included the composition of programs from modules, and Haskell and Clean just seem to have inherited this attitude towards programs from earlier generations of high-level languages. However, the paradigm shift involved here, between declarative and imperative programming on the language and operating system level, makes it obvious that the two parts of module handling take place on different abstraction levels: the often neglected task of building a working system from existing modules highlights a large hole in the programming abstraction provided by these modern functional languages which forces programmers to work on two different levels of abstraction.

Another source of problems is the static nature of the high-level part of Modula-style modules: they can be seen as source level structures, i.e., they structure the static representation of programs. This simple view of modules as source code structures completely fails to account for the specific properties of functional languages. Notably, a distinction between programs and expressions is introduced and programs (and thus program building blocks) are no longer first-class data objects in these languages. As an immediate consequence, the ability to abstract over programs as well as over data, which is an important feature of functional languages, is lost at this level of program construction. Contrast this with the level of programs as functional expressions, where it is just as easy to define program composition operators as it is to define data constructors, and if higher order functions are used to build programs, every language expression may function as a program building block. It is one of the main arguments in [Hug90] that this way of composing programs in functional languages is a significant contribution to modularity, and that:

> *The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. There-*

fore, to increase one's ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language. Complicated scope rules and provision for separate compilation help only with clerical details – they can never make a great contribution to modularization.

Hughes identifies higher order functions and lazy evaluation as two new kinds of glue provided by functional languages and concludes: 'Smaller and more general modules can be reused more widely, easing subsequent programming. This explains why functional programs are so much smaller and easier to write than conventional ones. It also provides a target for functional programmers to aim at. If any part of a program is messy or complicated, the programmer should attempt to modularize it and to generalize the parts. He or she should use higher-order functions and lazy evaluation as the tools for doing this.'.

Whereas the early publications on modular programming seemed to look at the program and especially at its source text as a part of the problem that had to be decomposed to be kept manageable, Hughes' work emphasized that *program composition* is at least as important as *problem decomposition*. He also demonstrated that modular programming in functional languages is **not** characteristic for programming-in-the-large, but applies equally well to all program sizes.

Functional languages do already provide sophisticated means for program composition out of small building blocks, and both the composition tools and the building blocks are simply expressions of the programming language. In contrast, conventional module systems are wrapped around the programming language in that programs are represented by a collection of static structures (modules) which contain the real program parts (declarations, statements, expressions, ...). Program parts can neither refer to programs nor to modules and the language is usually rather restricted and static at the level of modules, allowing only explicit references to other modules (import) and some control over the availability of program parts outside the module (export). Everything else is delegated to the programming environment, or rather, programmers are forced to take care of everything else at this level, which is usually inhabited by lots of fancy tools and ad hoc solutions, but little order. The lack of means for abstraction is only one example of this. In order to cope with the complexity of large systems, it seems quite natural to parameterize modules, i.e., abstraction over modules should be possible. But whether the principle of abstraction applies to modules depends on whether they can be described as a 'semantically meaningful syntactic category'. The common semantic description of modules are environments or collections of bindings which are not usually considered to be first-class objects of the programming language. On the other hand, the module system is usually not considered as a programming language in its own right and is thus too primitive to have an abstraction facility.

There is a good reason why the lack of explicit abstraction over modules need

not be obvious in conventional languages: the components of modules in these languages are procedure definitions, and each procedure is implicitly parameterized with the state of the global variable store. Therefore, a simple way to simulate parameterized modules is to pass the parameters as part of the variable state, i.e., modules may be parameterized by adjusting the contents of global box-variables. Of course, this trick is not easily available in purely functional languages and rightly so, as even in conventional languages its use will only lead to hidden dependencies between modules. However, if parameterized modules are not supported explicitly, the possibilities for modularization in comparison with imperative languages are severely compromised in these cases. Either the importing and the imported module become mutually recursive (the importing module depends on the import and the import depends on the parameter which is defined in the importing module) or each of the components of the imported module has to be parameterized separately (and consistently!).

To sum it up, Modula-style modules are sufficient for a large class of problems and can be added to functional languages, but this kind of module systems can hardly be called an adequate basis for modular programming in functional languages. However, before we look for more functional module systems, we continue our brief review of the historical development.

4.3 More highlights in the history of modular programming (1980 - today)

The predominant factor in the development of language support for modular programming since 1980 has been the type system. The precursor in this direction probably was the introduction of abstract data types as special modules or, as Liskov put it in her account of the history of CLU [Lis92]: 'I noticed that many of the modules discussed in the papers on modularity were defining data types'. Combined with Zilles' idea that 'a data type is completely characterized by the set of operations defined on the data' (see above), this describes the atmosphere in which more radical approaches to type systems could be pursued.

Probably the most radical idea was the notion of 'types as values', used in the design of Russell [DD85], Pebble [BL84] and Poly [Mat85] (among others). Russell's notion of data types was described as follows: 'A data type is a collection of named operations that provide an interpretation of values and variables of a single universal value space'. Type-checking was understood as checking the consistency of the interpretation, but more important for the present discussion is the fact that types, consisting of a finite set of named function values, were first-class values in Russell, providing for a very flexible form of modular programming based on data abstractions and *generic* or *polymorphic* procedures, i.e., procedures that can operate on values of more than one type.

While Russell preserved full static type-checking, Pebble took a different approach: 'We remove the sharp distinction between "compile time" and "run time", allowing evaluation (possibly symbolic) at compile time. This seems appropriate, given that one of our main concerns is to express the linking of modules and the checking of their interfaces in the language itself'. To this end, (collections of) bindings of variables to values were first-class values in Pebble, modules were modeled as collections of bindings and parameterized modules as functions from bindings to bindings. Through the use of *dependent types*, value bindings could depend on type bindings in the same collection, providing similar capabilities for user-defined types as Russell while separating the facilities used to build collections of bindings from those used for type formation. One of the motivations for the design of Pebble was to be able to formally address the part of the Cedar language (an extension of Mesa) which is concerned with data types and modules. The Pebble design incorporated many influential ideas, e.g., that 'the linking together of a number of modules ... should not be described in a primitive and ad-hoc linking language' but rather in a typed functional language. Another design objective was that a simple notation was needed 'for dealing with "big" objects (pieces of a program) as if they were "small" ones (numbers); this is the basic good trick in matrix algebra'. As a result, programming-in-the-large should look very much like programming-in-the-small.

While these activities were oriented towards experimental programming language design, they were accompanied by attempts to understand and develop type systems in general, and the type-theoretical basis of constructs for modular programming in particular. Mitchell and Plotkin [MP85] provided a framework in which representations of data abstractions were first-class data structures having *existential types*, i.e., if there is a representation implementing an abstract data type, then there exists a type on which the representation is based. The representation type is known only to the implementations of the abstract operations belonging to the particular representation, whereas, outside the representation, only the existence of such a type is known. Cardelli and Wegner [CW85] took up the idea in their more general survey of the notion of type in programming languages. They also showed how modules and data abstractions could be modeled using records with function components in an explicitly typed functional language called FUN and gave examples of possible applications. Although the authors noted that FUN could be the basis of a programming language, they used it solely to provide a 'framework for classifying and comparing existing languages and for defining new languages'. MacQueen [Mac86] discussed some shortcomings in the use of existential types for modular programming and investigated the use of dependent types instead.

MacQueen also designed the module language for Standard ML [Mac85], which soon reached the status of a reference system. It was not as radical as the other approaches described here (in particular, modules were not treated as first-class values), but it was freely available in a widely used general purpose

programming language and it was certainly among the most powerful module systems for this class. A series of type-theoretical accounts of the Standard ML module language and variants thereof appeared, addressing issues such as the stratified nature of its type system, the dependence of values in structures upon types in the same structure and the amount of information available on structures and functors. Whereas the early accounts focussed on a better understanding of the existing system, later ones also addressed modifications and extensions. Ultimately, this lead to proposals for higher order functors or even first-class structures.

Standard ML (SML for short) is a strongly typed functional programming language [HMM86, MTH90, MT91], and even though it also includes imperative features, its module language brings us back to the development of language support for modular programming in functional languages.

4.4 Towards functional module systems

Standard ML

The approach taken in the design of Standard ML was to take the module language seriously, as advocated earlier by DeRemer and Kron in [DK76]. However, the module interconnection language was not chosen to be essentially different from the programming language. Instead, the module language is a restricted functional language with modules (*structures* in SML terminology) as the basic data values [Mac85]. Interface specifications (*signatures*) serve as a kind of types for these values and first-order functions (*functors*) on modules can be defined and applied.

Structures in SML actually are encapsulated environments (collections of declarations), but look very similar to records. Qualified names are written like record selections and functors provide restricted means to handle structures. Indeed, in the implementation SML/NJ, which augments the module language of SML with higher order functors [CM94b], modules are coded as records and functors are coded as functions from records to records. Nevertheless, on the user level, there are two language levels with values, functions and types on the first and structures, functors and signatures (module interfaces) on the second level. There are two kinds of abstraction, two kinds of applications, and two kinds of types.

That a simple functional language can be used as a module interconnection language follows from the major requirements for such a language, as far as they have been discussed: the possibility of abstraction over modules and the availability of sophisticated means to compose programs from collections of modules. In spite of this, the SML approach uses two distinct functional languages for programming-in-the-small and programming-in-the-large. The module language

is introduced at an additional language level above the level of the programming language, which complicates the language design. While a module consists of (fragments of) functional programs, a functional program can still not talk about modules (they are not first-class values in SML). Furthermore, since the two language levels provide similar, but not equivalent constructs, users have to choose in which level they model their problem domains, with the module language being rather restricted compared with the programming language, but with some features that are only available on the module level (such as abstraction over types).

The decision to design SML as a two-level language had its origins in typing issues. Since structures are allowed to contain type declarations, a structure cannot have an ordinary type and thus cannot be an ordinary value [Mac85]. Recent research [MT94, CM94b, HL94, Ler94, Ler95] on the typing of structures and functors seems to aim at the elimination of this level distinction, i.e., providing first-class modules or at least higher-order functors without sacrificing static type inference (cf. also section 8.2).

Why modules?

The question seems to be counterintuitive at first, but arises naturally if we review some aspects of the historical development. There are Hughes' comments on modularity in functional languages. The Pebble designers have expressed their views that programming-in-the-large should not be too different from programming-in-the-small, and that the language to describe the linking of modules should be a functional one. And there is Standard ML with a module system that is based on these ideas, though using different functional languages for programming and program construction.

In functional languages, programs are first-class data objects, and it should not be necessary to divert programs to a second class status only to introduce program building blocks. On the contrary, if a language in which programs are first-class citizens does not support modular programming, there must be something fundamentally wrong with the definition used for first-class citizenship. The very essence of modular programming is the composition of programs from sub-programs that solve parts of the original problem, and all components of this basic scheme should be definable in functional languages: programs as problem-solving entities, programs as data objects for program composition, and even the task of program composition itself. So, instead of taking the need for an additional module system on top of the programming language for granted, we might better ask what is wrong with current functional languages. First of all, we need to know what exactly is missing in these languages that makes them unsuited for programming-in-the-large. Only then we can decide whether these shortcomings are unavoidable (making an additional level of program construction for modular programming necessary) or whether the current languages just need to be

completed in some respects.

Even among the early high-level languages, there were quite a few that allowed procedures or functions as data objects (including Lisp [McC60], Pal [AE68], Gedanken [Rey70] and Algol68 [Tan76]), but most of these languages did not approach the problem of program modularity directly, making an evaluation difficult. Morris [Mor73] used Gedanken to describe 'linguistic mechanisms which can be used to protect one subprogram from another's malfunctioning'. After mentioning the procedure as a means for abstraction, he claimed that 'in order to exploit this device to its fullest extent it is useful to make procedures full-fledged objects in the sense that they can be passed as parameters or values of other procedures and be assigned to variables'. Prompted by Morris' work, Zilles described in [Zil73] how his ideas about procedural encapsulation could be expressed in a language with support for function-returning functions. He used as an example the implementation of streams in the experimental operating system OS6 [SS72a, SS72b]: 'A stream is uniformly represented by a vector of entry points, one for each of the above operations. And, because the OS6 system language does not provide true function returning functions, the state information needed by these entries (...) is also stored in the vector.'.

Mesa was one of the first languages that had both modules and first-class procedures and, fortunately, the authors presented their early experience with the language in [GMS77]. After describing Mesa's module facility and the standard binding mechanisms (capable of simulating both Modula- and Simula-style modules), they made the following observations: 'Because Mesa has procedure variables, it is possible for a user to create any binding regime he wishes simply by writing a program that distributes procedures. Some users have created their own versions of Simula classes. They have not used the binding mechanism described above for a number of reasons. ... Their binding scheme deals with such situations by representing objects as record structures with procedure-valued fields. ... some fields of each record contain the state information necessary to characterize the object, while others contain procedure values that implement the set of operations. If the number of objects is much larger than the number of implementations, it is space-efficient to replace the procedure fields in each object with a link to a separate record containing the set of values appropriate to a particular implementation.'. The basic idea seems to have been borrowed from the OS6 implementation, but the experience report is surprising, because it means that in a language with both explicit language support for modules and first-class procedures, some users preferred to use records of procedures for modular programming.

In the early 1980s, Atkinson and Morrison reported on a similar attitude, this time from a language design perspective. Their paper was titled 'Persistent first class procedures are enough' [AM84] and described 'how the provision of a persistent programming environment together with a language that supports first class procedures may be used to provide the semantic features of other ob-

ject modeling languages'. They made these observations while working on the development of a persistent[1] variant of Algol [ABC+83b]. Influenced by the work of Morris and Zilles (to which they added the means to store procedures in a long-term storage), they described how 'the effects of information hiding, data protection and separate compilation are provided' in a practical programming language without explicit support for modules. They concluded: 'It has long been understood that it is desirable to be parsimonious in introducing concepts into a language design. The preceding demonstration therefore challenges language designers as to whether it is necessary to introduce a long list of concepts which can be covered by the persistent procedural mechanism.'. These ideas seem to have had absolutely no influence on the design of functional languages, but they have since been further developed outside the mainstream of functional programming research and have a strong relationship to our work which will be explained in more detail in section 8.3.

Later, the use of first-class functions in combination with records to model language constructs for modular programming became common practice in semantic and type-theoretic accounts (to name only a few: [Car84, CW85, KR94, GM94]). This use of records in theory seems to have been widely ignored as an option for practical language designs. This is partly due to typing problems (cf. section 8.2), but also due to a different interpretation of modules, namely that of encapsulated environments (bindings of variable names to values). While both records and environments can be used to model language constructs for modular programming in meta-languages, we cannot overemphasize the difference between these two views in practice: records are just heterogeneous data structures, and their use is relatively unproblematic, both in programming languages and in their meta-languages. In contrast, the meta-level concept of environments is closely coupled to the static binding structure of functional languages, and introducing environments as first-class data structures at the level of the programming language is therefore semantically more complex.

Informally, the problems result from two contrary objectives of language design: on the one hand, lexical scoping requires the variable names bound in any environment to be statically known, and on the other hand, flexible use of environments as a basis for modular programming requires all aspects of environments to be dynamic. In SML, this conflict is decided in favor of static scoping: structures (which are encapsulated environments) are simply denied a first-class status, e.g., there is no conditional expression in the module language. In extensions of SML's module language, elaborate static type systems mediate between static and dynamic aspects of environments, i.e., environments may be computed dynamically, but only those bindings which are statically known to exist in these dynamically computed environments are visible outside (cf. section 8.2 for references). The statically known type of environments (structure signatures in SML

[1]The notion of orthogonal persistence will be discussed in more detail in section 8.3

terminology) determines their binding structure. This approach is bound to lead to more restricted module systems as it would otherwise intertwine program execution (computing new modules), static typing (inferring module signatures) and static semantics (lexical scoping of identifiers) of the resulting language in unforeseeable ways.

In an attempt to provide a less complex type-theoretic framework for modular programming (compared with extensions of SML's module system), Jones [Jon95, Jon96] proposed a module system in which modules are modeled by first-class records structures. He claimed that his intermediate language 'is a useful and powerful language in its own right', and we interpret his work as an attempt to use records for modularization on the level of the programming language (the focus of his work is on the typing problems of this approach; cf. section 8.2).

Summary

The material reviewed in this chapter leads us to a slightly unusual conclusion about language support for modular programming. The problems experienced in programming-in-the-large stem from one simple fact: For large programs it becomes apparent that *the program itself has to be seen as data.* In particular, this data has to be organized into an adequate structure, and a language providing suitable abstractions and tools is needed to support the processes of program construction, reorganization, and maintenance. However, as long as the underlying programming language provides adequate means to handle large collections of data and makes no artificial distinctions between programs and data, there is no reason to bring a second language into play.

Functional programming languages do already treat programs as first-class data objects and therefore provide essential support for modular programming. If they lack any features needed to support programming-in-the-large, these shortcomings are likely to be related to the handling of large collections of data objects in general, i.e., the problems are not specific to modular programming. Still, the goal of better support for modular programming-in-the-large can serve to highlight these problems. First of all, large programs need to be organized into smaller units (program building blocks), and these units cannot always be functions but have to include collections of functions at least. To keep the first-class status of program building blocks, data structures of the programming language should be used to represent such collections (instead of external constructs such as modules), and records have already proven to be useful for this kind of task both in the theory of modular programming and in implementations. By copying records from the meta-level to the programming level, it should be possible to use the programming language as its own module language. With some further enhancements to the data representation and processing capabilities of the programming language, it should then be possible to organize the data represented by large programs in just the same way as any other large collection of data, which utilizes

the full expressive power of functional languages for the task of program construction. Furthermore, this will lead to orthogonal language extensions, which may be used not only for program management but for managing large collections of data in general.

Part II

The Language Framework

Chapter 5

Language Definition

The present chapter is the first of three chapters that present our language design. Its purpose is to formally define a functional programming language extended with facilities to specify modular program structures and interactions with run-time environments. The formal language definition should be understood as the interface[1] between programs written in the language (discussed in chapter 6) and implementations of the language (discussed in chapter 7). Since the foundations for the design decisions to be made have been explored in the first part of this thesis, the presentation is kept rather terse here. As for our major design goal of simplicity through generality, the simplicity of the language is directly reflected in the formal presentation, and the influence of the design principles of abstraction, correspondence, and data type completeness on design decisions is mentioned. However, the further discussion of our language design is postponed to chapter 6, where the three components of the language (functions, frames, and interactions) are combined in various ways to model several well-known approaches to modular programming. Some options for further work are discussed in chapter 8.

5.1 Notation and auxiliary definitions

Transformation systems are used for the formal description of all language parts to provide a uniform framework for the combination of context-free reduction rules and context-dependent interaction rules. However, in contrast to the transformation systems given in earlier chapters, not all rules are given directly in terms of the concrete syntax. Such an approach would be tedious for the more complex syntax of the language presented here, even if combined with abstraction over contexts: the language is defined as an extended λ-calculus, and many of the additional constructs do not differ from each other as far as the binding structure is concerned. Therefore, a naive definition of the basic language properties such as binding structure, substitution, and β-reduction for each language

[1] This notion is made more precise in section 6.4.

construct would be highly repetitive, hiding the essence of the definitions in a long list of similar transformation rules. As outlined in section 2.3, all additional constructs could be given definitions as λ-terms, but this would have the disadvantage of tying the operational behavior of the new constructs to that of the defining λ-terms. Such a dependency would undermine the idea of a high-level programming language as a layer of abstraction: to understand the constructs of the higher language level, one would have to think in terms of their definitions in a lower language level without these constructs.

The solution adopted here is to factor the terms of the concrete syntax into their basic properties, such as binding structure, subterms and an identity tag, thereby establishing a one-to-one correspondence between terms of the concrete syntax and terms of an abstract syntax. This factorization allows us to abstract from some of the properties and thus to reason independently about binding structure and transformation rules. In particular, basic language properties can be defined without reference to the concrete language syntax, which also simplifies the introduction of additional language constructs: a concrete syntax of the new constructs is defined to extend the language syntax, a mapping into the abstract syntax is defined to extend the basic language properties to the new constructs, and finally, extension-specific transformation rules may be defined to extend the language semantics. This approach allows for a concise presentation of the formal language definition and for a clear separation between different aspects of the language.

$\mathcal{FORM}^+$	:	non-empty sequences of $\mathcal{FORM}$
$\mathcal{FORM}^*$	:	possibly empty sequences of $\mathcal{FORM}$
$\mathcal{FORM}_1 \ldots \mathcal{FORM}_n$	:	an individual sequence of $\mathcal{FORM}$
$(\mathcal{FORM}_i)_{1 \leq i \leq n}$	:	
$(\mathcal{FORM}_i)_{i \in \{1,\ldots,n\}}$	:	the same sequence as an indexed family
$\vec{v}$	:	an abstract sequence named v (with elements v_i)
$\#_x \ \vec{v}$	:	the number of occurrences of x in the sequence v

Figure 5.1: Notational conventions for sequences of syntactic forms

Sequences of syntactic forms occur frequently in the definitions, e.g., to specify lists of variables or expressions, and so we introduce some notational conventions for sequences in figure 5.1. The use of $^+$ for non-empty sequences and of * for possibly empty sequences is common in syntax definitions, whereas the notation for indexed families and vectors is borrowed from mathematics. The indexed family notation is particularly useful when it is possible to abstract from the index set, or when an operation is applied uniformly to all elements of a sequence, and the vector notation is used in definitions where not even the individual elements of sequences are important.

The purpose of the concrete language syntax is to provide a readable representation, while the abstract language syntax suppresses syntactical details and provides uniform access to those features of language constructs that are relevant to the formal presentation. These are (a) the distinction between elementary constructs and those composed from sub-expressions, (b) the distinction between complex constructs that establish a new scope for variables and those that do not, and (c) the identity of individual constructs. The general form of an abstract language construct is

$$Con^n_{\vec{v}}\ e_1 \ldots e_n$$

where n is the arity of the construct, $e_1 \ldots e_n$ are the sub-expressions (if $n = 0$, the construct is a constant), and the elements of $\vec{v}$, denoted by v_j, are the variables that are bound only in the sub-expressions. Con itself is a tag that identifies the syntactical construct, providing for a one-to-one mapping between concrete and abstract syntax. Note that the concrete syntax may be further enriched with syntactic sugar, making additional labels necessary in the abstract syntax.

$\forall x \in var,\ \vec{v} \in var^+,\ n,k,m \in Nat,\ i,j \in \{1,\ldots,k\},\ Con \in Tags,\ M,N \in AS$:

$$\begin{array}{rcl} AS & = & \backslash^n\ var \\ & | & Con^k_{\vec{v}}\ (AS_j)_{j\in\{1,\ldots,k\}} \end{array}$$

Figure 5.2: Abstract syntax

$$\begin{array}{rcl} BC^0_x & = & [\ ] \\ BC^{n+m}_x & = & Con^k_{\vec{v}}\ (AS_j)_{1\le j<i}\ (BC^n_x)\ (AS_j)_{i<j\le k}\ ,\ if\ m = (\#_x\ \vec{v}) \end{array}$$

$$\text{the occurrence of x in } BC^n_x[\ \backslash^k\ x\ ] \text{ is } \begin{cases} bound & if\ k < n \\ free & if\ k = n \\ protected & if\ k > n \end{cases}$$

Figure 5.3: Binding structure

Using this abstract form of language constructs (defined in figure 5.2), it is possible to outline the basic properties common to certain classes of language constructs before any concrete constructs are introduced. Figure 5.3 defines the binding structure, using binding contexts BC, and figure 5.4 adds definitions for substitution and variable protection. The only concrete language element needed

for these definitions are protected variables as introduced in section 2.2, which are therefore included in the abstract syntax. Note that the generalized definitions of binding contexts, substitution and variable protection closely resemble those given for the λ-calculus (figures 2.2, 2.3 and 2.4 in section 2.2) – they just abstract away syntactical detail and are also generalized to handle constructors binding multiple variables. When reading the formal definitions in figure 5.4, it is useful to recall the intended meanings of the auxiliary operations: $(\Pi^m_{x,n}\ M)$ modifies the number of protection keys of each at least n-fold protected occurrence of x in M by m, and $([\backslash^n\ x \leftarrow N]\ M)$ substitutes N for each n-fold protected occurrence of x in M. The formal definitions for multi-variable protection $(=_{\vec{\Pi}})$ and substitution $(=_{\vec{\sigma}})$ are complicated because of the need to prohibit interactions between the individual substitutions (so that the multi-variable substitution is not defined as a sequence of cumulative substitutions but as a set of independent substitutions). However, this complexity is local to the definitions and the use of these operations is therefore simplified.

$\forall x, y \in var,\ \vec{v} \in var^+,\ M, N, M_i, N_i \in term,\ n, k, j, r \in Nat,\ m \in Int:$

$$\begin{aligned}
\Pi^m_{\vec{v}x,n}\ M &=_{\vec{\Pi}} \Pi^m_{\vec{v},n}\ \Pi^m_{x,n}\ M \\
[(v_i \leftarrow N_i)_{1\le i\le n}]\ M &=_{\vec{\sigma}} \Pi^{-1}_{\vec{v},1}\ ([v_i \leftarrow \Pi^{+1}_{\vec{v},0}\ N_i])_{1\le i\le n}\ \Pi^{+1}_{\vec{v},1}\ M
\end{aligned}$$

$$\begin{aligned}
\Pi^m_{x,n}\ \backslash^k x &=_{\Pi} \backslash^{k+m} x,\ if\ k \ge n \\
\Pi^m_{x,n}\ \backslash^k y &=_{\Pi} \backslash^k y,\ if\ (k < n)\ \vee (y \ne x) \\
\Pi^m_{x,n}\ (Con^k_{\vec{v}}\ (M_i)_{1\le i\le k}) &=_{\Pi} Con^k_{\vec{v}}\ (\Pi^m_{x,n+r}\ M_i)_{1\le i\le k},\ if\ (r = (\#_x\ \vec{v}))
\end{aligned}$$

$$\begin{aligned}
[\backslash^n x \leftarrow N]\ \backslash^n x &=_{\sigma} N \\
[\backslash^n x \leftarrow N]\ \backslash^j y &=_{\sigma} \backslash^j y\ ,\ if\ (j \ne n) \vee (y \ne x) \\
[\backslash^n x \leftarrow N]\ (Con^k_{\vec{v}}\ (M_i)_{1\le i\le k}) &=_{\sigma} Con^k_{\vec{v}}\ ([\backslash^{n+r} x \leftarrow \Pi^{+1}_{\vec{v},0}\ N]\ M_i)_{1\le i\le k}\ , \\
&\qquad if\ (r = (\#_x\ \vec{v}))
\end{aligned}$$

Figure 5.4: Substitution of expressions for free variables

Given these general definitions, the basic properties of individual constructs are clear from their abstract representations (this classification is the main use we make of the abstract syntax in later sections; specific transformation rules are much more readable in terms of concrete syntax). The approach taken here separates the issues of substitution (binding structure, variable protection) from β-reductions and allows the introduction of new binding constructs with individual reduction rules. In general, these additional reduction rules deviate from

those that would result from a translation into the λ-calculus only in so far as they are defined directly on the new constructs. For instance, what would have been several steps in a λ-representation is defined as one large step, and the results are expressed in terms of high-level constructs again. Accidental reductions (that would not conform to the high-level view of the constructs in question, but would be possible if high-level constructs had been mapped to a λ-representation) are also ruled out.

5.2 The functional part

The focus of the language definition presented in this chapter is on the extensions and on the interactions between the three language parts (functions, frames, and interactions). The constructs of the functional core (cf. figure 5.5) are fairly standard for an extended λ-calculus, including multi-parameter abstractions and applications, local (recursive) definitions, numbers, booleans, strings, lists and user-defined data constructors (starting with an upper-case letter to distinguish them from keywords and variable identifiers, which start with lower-case letters) with the usual primitive operations on these data types. Not all parts of the given reduction language have been included (for instance, pattern matching is omitted to simplify the presentation) but it should be clear that the core described here corresponds to a subset of a practical programming language. The language is derived from an untyped λ-calculus, but the notion of types was implicitly introduced when constants and primitive operations were added to the language. No formal definitions of the primitive operations are presented here but, similar to the conditional (cf. figure 5.6), they all apply only to certain types of expressions. Nevertheless, β-reduction and substitution are still defined on all language expressions, and types are not explicit on the language level. Therefore, we sometimes refer to the language as implicitly and dynamically typed, even though the λ-calculus part is still untyped. Since we do not make any unusual assumptions about features of the core language, any functional language which subsumes the λ-calculus and is not restricted in itself, e.g., by constraints of a static type inference system, could be used as the functional core of our language.

The concrete syntax of the functional core language is given in figure 5.5. Those language constructs that have a non-trivial sub-structure with respect to the abstract syntax are annotated with their abstract forms. As an example, the `let`-construct has as sub-expressions the right-hand sides of the local definitions and a body expression, and the variables defined in the definition part of the `let`-construct are bound only in this body expression, so two abstract constructs are needed: LET holds the right-hand sides of the local definitions and the auxiliary construct $LETH$, but introduces no variable bindings, and $LETH$ holds the body and binds the local variables. In contrast, the local variables in the `letrec`-construct are bound both in the right-hand sides of the definitions and

in the body, hence only one abstract construct ($LETREC$) is needed. The only other language construct that introduces local variable scopes is λ-abstraction ($LAMBDA$).

$$
\begin{array}{lll}
\forall n, k, i \in Nat,\ v \in var,\ e_i \in expr,\ ,\ \vec{v} \in var^+ : & & \\
expr & =_S & \backslash^n\ var \qquad\qquad\qquad\qquad : \backslash^n\ var \\
 & | & const \\
 & | & op \\
 & | & (expr\ (expr_i)_{1 \le i \le k}) \qquad : APPLY^{k+1}\ expr\ (expr_i)_{1 \le i \le k} \\
 & | & \lambda \vec{v}.\ expr \qquad\qquad\quad : LAMBDA^1_{\vec{v}}\ expr \\
 & | & \texttt{let}\ (v_i = def_i)_{1 \le i \le k} \\
 & & \texttt{in}\ expr \qquad\qquad\quad : LET^{k+1}\ (def_i)_{1 \le i \le k}\ (LETH^1_{\vec{v}}\ expr) \\
 & | & \texttt{letrec}\ (v_i = def_i)_{1 \le i \le k} \\
 & & \texttt{in}\ expr \qquad\qquad\quad : LETREC^{k+1}_{\vec{v}}\ (def_i)_{1 \le i \le k}\ expr \\
 & | & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \quad : COND^3\ expr_1\ expr_2\ expr_3 \\
 & | & <(expr_i)_{1 \le i \le k}> \qquad\quad : LIST^k\ (expr_i)_{1 \le i \le k} \\
op & =_S & primOp \\
 & | & Konstr \\
primOp & =_S & arithOp \\
 & | & relationalOp \\
 & | & listOp \\
 & | & stringOp \\
const & =_S & char \\
 & | & string \\
 & | & bool \\
 & | & num
\end{array}
$$

Figure 5.5: Concrete and abstract syntax of the core language

Figure 5.6 presents the reduction rules of the functional core language. The reduction rules include the λ-calculus rules α, β, and η, but also specific rules for conditional expressions and for local (recursive) definitions. Unless explicitly stated otherwise, all rules are context-free, i.e., they apply in all reduction contexts RC[2]:

$$
\begin{array}{l}
\forall Con \in Tags,\ k \in Nat,\ 1 \le i \le k,\ v \in var^* : \\
\qquad\qquad RC \ = \ [\ ] \mid Con^k_{\vec{v}}\ (AS_j)_{1 \le j < i}\ (RC)\ (AS_j)_{i < j \le k}
\end{array}
$$

The rules for conditional expressions are an example of δ-rules: they select one of the alternatives (the expressions following `then` and `else`) and apply only if the condition can be reduced to one of the constants `true` and `false` (otherwise, the

[2]We regard the choice of a specific reduction strategy as an implementation decision (cf. chapter 7).

$\forall M, N, body_i, body \in expr, \vec{M} \in expr^+, \vec{v} \in var^+, x, y, v_i, p_i \in var, i, j \in Nat$:

$$\begin{array}{rcl} \lambda v\ \vec{v}.M & =_S & \lambda v.\lambda \vec{v}.M \\ (M\ N\ \vec{M}) & =_S & ((M\ N)\ \vec{M}) \\ v_i\ \vec{p} = M & =_S & v_i = \lambda \vec{p}.M \end{array}$$

$$\begin{array}{rcl} \lambda x.M & =_\alpha & \lambda y.\Pi_{x,1}^{-1}\ [x \leftarrow y]\ \Pi_{y,0}^{+1}\ M \\ (\lambda x.M\ N) & =_\beta & \Pi_{x,1}^{-1}\ [x \leftarrow \Pi_{x,0}^{+1}\ N]M \\ M & =_\eta & \lambda x.(\Pi_{x,0}^{+1}\ M\ x) \end{array}$$

$\texttt{if true then}\ M\ \texttt{else}\ N$
$\quad =_\delta \quad M$
$\texttt{if false then}\ M\ \texttt{else}\ N$
$\quad =_\delta \quad N$
$\texttt{let}\ (v_i = body_i)_{i \in I}\ \texttt{in}\ body$
$\quad =_\beta \quad [(v_i \leftarrow body_i)_{i \in I}]\ body$
$\texttt{letrec}\ (v_i \vec{p_i} = body_i)_{i \in I}\ \texttt{in}\ body\ ,\ if\ body \notin \{v_i \mid i \in I\}$
$\quad =_\rho \quad [(v_i \leftarrow \texttt{letrec}\ (v_i \vec{p_i} = body_i)\ \texttt{in}\ v_i)_{i \in I}]\ body$
$\texttt{letrec}\ (v_i \vec{p_i} = body_i)_{i \in I}\ \texttt{in}\ v\ ,\ if\ (j \in I) \wedge (v = v_j)$
$\quad =_\rho \quad [(v_i \leftarrow \texttt{letrec}\ (v_i \vec{p_i} = body_i)\ \texttt{in}\ v_i)_{i \in I}]\ \lambda \vec{p_j}.body_j$

Figure 5.6: Reduction rules of the core language

construct is syntactically valid but irreducible). The rule for `let` is just a special instance of β-reduction, but the rules for `letrec` are new high-level rules that allow collections of mutually recursive definitions to be substituted in two steps: first, every occurrence of recursively bound variables in the body expression is substituted by a duplicate of the recursive definitions (with the recursively bound variable as body expression), and then the appropriate right-hand side is selected from the definitions (with all occurrences of recursively bound variables in this right-hand side substituted by a duplicate of the recursive definitions). Note that recursively bound variables are never substituted in the right-hand sides of the recursive definitions themselves, while reduction in all sub-expressions of these constructs is allowed (by their translation into the abstract syntax and the definition of reduction contexts).

5.3 Interactions with runtime environments

The survey of current input/output-frameworks for functional languages in chapter 3 left only one basic design decision open: should the environment objects have explicit representations on the language level, in which case an environment passing style with static uniqueness typing [AP95] would be the way to go, or should the environment remain implicit, in which case a monadic style of input/output-programming [PJW93] would be the most flexible alternative? The fact that the constructs of a monadic style of input/output could be easily defined in a uniqueness typed environment passing framework seems to suggest that the environment passing style is more fundamental, but the question is: fundamental to what? Uniqueness typing addresses the more general problem of statically controlled use of resources in functional programs and, even if combined with passing unique representations of environment objects as arguments to these programs, it does not suffice to solve the input/output-problem. The reason is that the environment is not updated in one conceptual step *after* the evaluation of a program has computed a new environment representation but rather in small steps *whenever* the environment representation is modified during program evaluation. The primitive interactions are thus implemented as side-effecting operations, the use of which is rendered safe in the uniqueness-typed environment passing framework.

Similarly, monads are used to address the more general problem of computations (involving state, input/output, backtracking, ...) returning values: they do not solve any input/output-problems directly but rather provide an elegant and flexible abstraction of many solutions to related problems [Wad92b]. In chapter 3, we have developed monadic input/output-constructs simply as means to communicate with an external resource manager for the program's runtime environment. That these constructs can be viewed as an instance of the monad concept emphasizes the importance of this concept: it helps to embed the input/output-

constructs into larger frameworks based on other monads, and hence contributes to the seamless integration of imperative language features into the functional world. However, it does not solve the basic problems of input/output. For instance, no less than three different input/output-schemes are used to solve these basic problems in [PJW93], the paper which originally proposed 'a new model, based on monads, for performing input/output in a non-strict, purely functional language'. On the user level, an abstract data type of interaction scripts is provided (cf. section 3.4.2), which are represented internally by state-transforming functions, i.e., the user program is expanded to a functional program based on environment passing. This intermediate program representation is generated from user programs, which do not have direct access to the environment representation (called 'world'), in such a way that it is guaranteed to use the 'world' in a single-threaded way. The intermediate program representation is used to keep the input/output-handling correct during optimizing program transformations. Finally, the optimized functional program is compiled to imperative code where the order of execution is encoded in the order of statements, and the explicit 'world' can be discarded. This final representation is based on side-effects, but is faithful to the original program by construction.

So, both input/output-schemes merely provide frameworks in which side-effecting operations can safely be used with a guaranteed order of execution and without affecting the properties of the purely functional parts of the language. Currently, we do not think that one of these two frameworks is fundamentally better suited for the problem of input/output than the other. Both have distinct advantages over each other depending on the problem domain, but most of these differences are not relevant to the problem at hand.[3] However, a monadic style, without the translations into other input/output-schemes proposed in [PJW93], seems to require fewer and smaller modifications to the core language than an environment passing style. Especially the static uniqueness type checking on which the latter style is based runs counter to the idea of untyped β-reduction and substitution. It would be possible to relax the uniqueness condition a bit, checking it dynamically and only when interactions are about to be performed, but this would still leave a non-local dependence of interactions on the number of environment copies in the whole program. Finally, for reasons explained in chapter 3, it would make sense to use a monadic style of input/output even in an environment passing framework.

For our current language design, we choose to base interactions on constructs similar to those developed in section 3.4.2, which seem to provide exactly the right level of abstraction for our input/output-problem and happen to form a monad,

[3]They immediately became relevant if we wished to extend the input/output-framework to achieve non-sequential evaluation. By splitting up the environment into multiple independent parts, environment passing extends easily to multiple threads of computation inside the evaluation of one program [Ach96], whereas the monadic style extends just as easily to multiple programs sharing one environment [PGF96].

too. In other words, there is a type of interaction scripts, a binary operator `>>=` to compose scripts with continuation functions, and a unary operator `return` to act as an identity interaction and to carry the results of intermediate interactions. Primitive interactions, `>>=`, and `return` are the only constructors of interaction scripts, and interactions are only performed in specific interaction contexts to ensure a deterministic sequence of transformations of the environment state (cf. figure 5.7). The next design question is which assumptions should be made about the runtime environment, and we opt for an abstract variant of current file systems (UNIX file systems in particular): the environment is assumed to consist of named files, where each file contains a sequence of file elements. As the UNIX example shows [RT83], other devices such as communication streams and terminals are easily mapped into such a model. Finally, we need to decide about the primitive interactions, and we choose to discuss here only the very basic interactions, namely to get objects from files and to put them there.

Inside this framework for input/output, a couple of secondary issues arises: how can the idea of get and put interactions be integrated with the abstract file system, and what exactly are the objects of interactions? As for the first issue, get and put interactions need to address the files they operate on, and a simple file name parameter would barely suffice for this purpose. For instance, files persist in the file system, but without any means to redirect input interactions to the beginning of files after they have been read, each file could only been read once in each program. We adopt a standard solution and introduce one level of indirection: before files can be accessed with get and put interactions, local connections have to be established to these files via *file handles*. Essentially, each file handle represents a stream connected to a file (or to any other device that can be mapped into the file system), and after it is created, interactions proceed sequentially through the contents of the file connected to the handle.

The issue of interaction objects has been avoided in chapter 3 by restricting the attention to character-based input/output. Indeed, this is the common idea of interaction objects not only in functional languages (with slight variations that allow bytes and numbers to be communicated). However, this view is in immediate conflict with the principle of data type completeness, as it selectively restricts the rights of more complex data objects. For simple data structures, this restriction is merely a matter of convenience, and programmers could take up the tedious and error–prone task of providing explicitly programmed two-way mappings between their data structures and sequences of characters. For some types of expressions, however, it is not possible for programmers to provide such a mapping, and these types include the types of functions. The restricted view adopted in many common functional languages is therefore not only in conflict with one of our chosen language design principles but also in sharp contrast to the claim of having functions as first-class citizens in these languages. From a language design perspective, there is no reason for such restrictions. Therefore, we choose to abandon the restriction of input/output to characters and allow

all valid language expressions to be communicated via interactions. This unrestricted view also provides input/output-operations at a higher level of abstraction, because programmers do not need to deal with the details of communicating sub-structures of complex data structures or even with representation conversions from and to sequences of characters.

$$
\begin{array}{lll}
op & =_S & primInter \mid \texttt{>>=}^2 \mid \texttt{return}^1 \\
primInter & =_S & \texttt{fput}^2 \mid \texttt{fget}^1 \mid \texttt{fopen}^1 \mid \texttt{fclose}^1
\end{array}
$$

$$
IC \;=\; [\,] \mid (IC \texttt{ >>= } expr)
$$

Figure 5.7: Syntax extensions for interactions, interaction contexts

Since the basic design issues are settled, it is now possible to formally define the interaction capabilities of our language. The extensions to the language syntax are introduced in figure 5.7, together with a definition of interaction contexts IC). The additional identifiers of primitive interactions extend the syntactic category *op* (the operators are annotated with their arities, and we assume that `>>=` is always used as an infix operator). The formal description of interactions between programs and their runtime environments is a bit more involved than the description of program transformations. The reason is that every interaction comprises two different kinds of transformations that are meant to take place in one conceptual step. Both transformations of the environment state and program transformations need to be described with one set of transformation rules over objects that have to represent both the current program and the current state of the environment. This also means that contexts for program transformations have to be stated explicitly, and that the program transformations involved in interactions are context-sensitive: not only do the program contexts have to be restricted to interaction contexts (IC) to guarantee a well-defined order of execution for the primitive interactions, but the results of these interactions do also depend on the part of the context that models the current state of the environment.

The formal means to describe general interactions have been developed in chapter 3. In order to formally define the primitive interactions in figure 5.8, the environment state is modeled as a mapping $((h \to obj_h)_{h \in H})$ from *handles* to *objects*. Handles can be compared for equality and include *file names* and *temporary file handles*. File names are mapped to *Files*, which are modeled as indexed sequences of *file elements* and can be accessed via temporary handles. The interaction `fopen` is used to open a new stream to a file. If the file does not exist, it is created as an empty sequence of file elements (second rule), otherwise only a new file handle is generated (first rule). The file handles that are returned to the program are mapped (in the environment) to pairs of file positions and file names, i.e., each file handle allows access to one file at a handle-specific

$$IC[\ (\texttt{fopen}\ name)\ ]\ ||\ (h \to obj_h)_{h\in H}\ ,\ if\ (name \in H) \wedge (handle \notin H)$$
$$=_I\ \ IC[\ (\texttt{return}\ handle)\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (0, name)) \end{pmatrix}$$
$$IC[\ (\texttt{fopen}\ name)\ ]\ ||\ (h \to obj_h)_{h\in H}\ ,\ if\ (name \notin H) \wedge (handle \notin H)$$
$$=_I\ \ IC[\ (\texttt{return}\ handle)\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (0, name)) \\ (name & \to & ()) \end{pmatrix}$$
$$IC[\ (\texttt{fclose}\ handle)\ ]\ ||\ (h \to obj_h)_{h\in H}\ ,\ if\ handle \in H$$
$$=_I\ \ IC[\ (\texttt{return 'fclose'})\ ]\ ||\ (h \to obj_h)_{h\in H\setminus\{handle\}}$$
$$IC[\ (\texttt{fput}\ handle\ expr)\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (i, name)) \\ (name & \to & (elem_j)_{j\in J}) \end{pmatrix}$$
$$=_I\ \ IC[\ (\texttt{return 'fput'})\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (i+1, name)) \\ (name & \to & (elem_j)_{j\in J\setminus\{i\}}\ (expr)_{j=i}) \end{pmatrix}$$
$$IC[\ (\texttt{fget}\ handle)\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (i, name)) \\ (name & \to & (elem_j)_{j\in J}) \end{pmatrix}\ ,\ if\ \begin{matrix} (i \in J) \wedge \\ (elem_i \in expr) \end{matrix}$$
$$=_I\ \ IC[\ (\texttt{return}\ elem_i)\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (i+1, name)) \\ (name & \to & (elem_j)_{j\in J} \end{pmatrix}$$
$$IC[\ (\texttt{fputc}\ handle\ char)\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (i_c, name)) \\ (name & \to & (char_j)_{j\in J}) \end{pmatrix}$$
$$=_I\ \ IC[\ (\texttt{return 'fputc'})\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (i_c+1, name)) \\ (name & \to & (char_j)_{j\in J\setminus\{i\}}\ (char)_{j=i_c}) \end{pmatrix}$$
$$IC[\ (\texttt{fgetc}\ handle)\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (i_c, name)) \\ (name & \to & (char_j)_{j\in J}) \end{pmatrix}\ ,\ if\ (i_c \in J)$$
$$=_I\ \ IC[\ (\texttt{return}\ char_{i_c})\ ]\ ||\ \begin{pmatrix} (h & \to & obj_h)_{h\in H} \\ (handle & \to & (i_c+1, name)) \\ (name & \to & (char_j)_{j\in J} \end{pmatrix}$$

$$IC[\ ((\texttt{return}\ N)\ \texttt{>>=}\ M)\ ]\ \ =_I\ \ IC[\ (M\ N)\ ]$$

Figure 5.8: Interaction rules

position. The interaction `fclose` has to be called explicitly to remove temporary file handles (and thus to close a connection to a file). The interaction `fput` stores an expression in a file (at a position indicated by a file handle), and the interaction `fget` retrieves one expression from a file (again via a file handle). Rules for character-based variants `fgetc` and `fputc` are also given, but not further discussed here. The rules for the primitive interactions simply formalize input and output, either character-wise (`fgetc,fputc`) or expression-wise (`fget,fput`), modifying file indices and contents accordingly, and only one additional rule (the last in the figure, for `>>=`) is needed to describe how intermediate results are passed on to continuations. In combination with the use of interaction contexts IC (defined in figure 5.7), these rules model a depth-first traversal of tree-shaped interaction descriptions, following the ideas described in section 3.4.2.

For readers familiar with monads, it might be strange to see the rule for composition of interaction descriptions (`>>=`) in figure 5.8 restricted to interaction contexts. It is one of the monad axioms (`return` as left identity to `>>=`) and should be applicable to all continuations that return interaction descriptions. However, our language is not restricted to allow for static type checking, and it is, in general, not decidable whether or not the application of the expression in continuation position to the expression in intermediate result position will reduce to an interaction script. As long as it does, there is no difference between the rule given in figure 5.8 and a variant with general contexts instead of interaction contexts, because the resulting interaction script could only be evaluated in an interaction context anyway. It is thus safe to avoid the general form of the rule, and it is reasonable, too, for the monad axioms do only apply to expressions of certain types.

There are several other things to note here. First, files in the environment are modeled as sequences of elements, leaving open the definition of elements. Since the environment stands for entities outside the language definition (such as file systems), we can hardly be more specific here. Even so, we have to make some assumptions, e.g., of sequential access to files, to model common features (such as files and communication channels). These assumptions should be read as prerequisites that need to be established in the actual environment in order for the interaction rules to be applicable, i.e., the assumptions describe the interface of the language definition to an environment definition. This includes the assumption in the general `fget/fput`-rules that every valid language expression or a suitable representation thereof can become an element of a file, and the assumption in the specific `fgetc/fputc`-rules that any file of elements can also be viewed as a file of characters. The latter assumption corresponds to the view of files as sequences of bytes or characters which is common in UNIX environments while the former assumption represents our view that there should be no unnecessary restrictions on the type of expressions that can be used in input/output-operations. Second, only few primitive interactions are given here. The reasons for this are that we are concerned with a presentation of the general

framework here, for which the selected subset of interactions suffices, and that the complete set of interactions provided depends on issues specific to individual languages and environments. For the extension of our reduction language, we chose primitive interactions similar to those in a UNIX/C environment [KR88] with some minor adaptations (the character-based part of these is described in [Tim96]). However, this was merely seen as an intermediate solution, proven to be useful in practice, to collect some experience with input/output in a functional language before designing a dedicated set of primitives (cf. chapter 8.1).

Finally, there are some pragmatic considerations that force us to refine the rules given so far: the primitive interactions differ from other primitive operations (δ-rules) in that they depend on an invisible parameter (we assume that only the program will be directly visible to programmers). If the δ-rules do not allow to reduce an application of a primitive function, the application simply remains constant: it is an equivalent and irreducible form of the original program, and programmers find in it detailed information on why their original program is not further reducible, e.g., because a primitive operation is applied to arguments of the wrong type or because of the use of a nonexistent index in a selection operation from a list. They can thus identify the problems and modify their original program accordingly. However, if a primitive interaction is not executable, this may have two possible causes, being either related to the explicitly given parameters or to the implicitly given state of the environment. Since we have chosen an input/output-framework in which the environment is implicit and external to the program, it may not be sufficient to present a non-executable primitive interaction to programmers, as they would not be able to identify problems related to the current state of the environment.

Therefore, we choose to execute primitive interaction in two phases: first, the applicability of interaction rules is checked with respect to the explicit parameters, and if there is no applicable rule in this first phase, the interaction remains constant just as if was a primitive function. If the set of potentially applicable rules found in this first phase is not empty, execution of the interaction is *attempted*, and this attempt may either succeed or fail, depending on the current state of the environment. The former case is described by the rules in figure 5.8, where it is only necessary to change the `(return expr)`-parts on the right-hand side of the rules to `(return (OK expr))` to distinguish these results from the second case, in which the primitive interaction is replaced by `(return (NE` *message*`))` and the environment state is left unaltered. `OK` and `NE` are data constructors holding the results of successfully executed and non-executable interactions, respectively, and *messages* are strings describing the problems. Programs may thus provide continuations for both possible outcomes[4]. To this end, a `check`-primitive is included in the language subset described here[5] to dispatch the interaction result

[4]additionally, there should be primitive interactions to query the state of the environment

[5]in the full language, this primitive can be defined in terms of the pattern matching facilities,

to a success or to a failure continuation, depending on the constructor holding the result (cf. figure 5.9).

$\forall M, N, P \in expr$:

$$\begin{array}{lll} \texttt{check}\ M\ N\ (\texttt{OK}\ P) & =_\delta & (M\ P) \\ \texttt{check}\ M\ N\ (\texttt{NE}\ P) & =_\delta & (N\ P) \end{array}$$

Figure 5.9: Dispatch of interaction results

5.4 Modular programming

The final part of the language design is concerned with support for modular programming. Here we decide not to add a module language on top of the programming language, but to use the programming language itself as its own module language. This decision is based on the survey of language support for modular programming in chapter 4 and on the design principle of simplicity through generality. The reasons for not using a separate module language are the additional complexity introduced by a stratified design and the search for elementary and general language constructs. Furthermore, we do not want complex constructs composed of several features (such as modules with import and export control, or classes with built-in inheritance) because these would predefine the ways in which the features could be composed and would force us to introduce various complex constructs where simple recombinations of a few elementary constructs should do. We are convinced that some of the elementary constructs needed for modular programming are already available in our language, and that those constructs that need to be added can be used for other purposes, too. The survey revealed collections of language expressions accessible by name and abstraction as the basic tools for modular programming. Support for abstraction is one of our design principles and provided in our language by λ-abstractions (cf. also section 6.4 for a more detailed discussion of this aspect), and collections of expressions are simply data structures. Since access to the structure components should be by name, record-like structures are the obvious choice. Alternatives would have been first-class environments, i.e., collections of bindings, using variable names to name components, or finite functions mapping names to components. The former would have incurred either the risc of losing static scoping or the introduction of complex constraints to avoid this (cf. the discussion in section 4.4), and the latter would have unnecessarily overloaded the available means to construct functions using λ-abstractions.

which are neither described nor used here.

$$
\begin{array}{lcll}
expr & =_S & frame & \\
primOp & =_S & .^2 \mid \texttt{delete}^2 \mid \texttt{update}^3 & \\
 & \mid & \texttt{test}^2 \mid \texttt{slots}^1 & \\
frame & =_S & \{slot^*\} & : \; FRAME^n \; slot_i \\
slot & =_S & string::expr & : \; SLOT^2 \; string \; expr
\end{array}
$$

Figure 5.10: Syntax extensions for frames

Record-like data structures are named *frames* here for historical reasons. The syntax for frames is kept simple (cf. figure 5.10): a frame is a (possibly empty) sequence of *slots* where each slot is a pair of *slotname* and *slotvalue*. A slotname is just a string (and thus a data object, not a variable identifier), and a slotvalue can be any syntactically valid expression. Note that frames are expressions and share all features of expressions, e.g., frames can be written to files and read from there using `fput` and `fget`, respectively, and frames can be parameters and results of functions.

$\forall s, s_i \in string,\ e, e_i \in expr,\ j \in Nat$

$$
\begin{array}{rcl}
(\{(s_i::e_i)_{i\in I}\} \; . \; s) & =_\delta & e_j \;,\; if \; (j \in I) \wedge (s_j = s) \\
(\texttt{delete} \; s \; \{(s_i::e_i)_{i\in I}\}) & =_\delta & \{(s_i::e_i)_{i\in I, s_i \neq s}\} \\
(\texttt{update} \; s \; e \; \{(s_i::e_i)_{i\in I}\}) & =_\delta & \{(s_i::e_i)_{i\in I, s_i \neq s} \; (s::e)\} \\
(\texttt{test} \; s \; \{(s_i::e_i)_{i\in I}\}) & =_\delta & \texttt{true} \;,\; if \; \exists i \in I : s_i = s \\
(\texttt{test} \; s \; \{(s_i::e_i)_{i\in I}\}) & =_\delta & \texttt{false} \;,\; if \; \forall i \in I : s_i \neq s \\
(\texttt{slots} \; \{(s_i::e_i)_{i\in I}\}) & =_\delta & < (s_i)_{i\in I} >
\end{array}
$$

Figure 5.11: Reduction rules for frames

The primitive operations on frames are essentially those of extensible records as proposed in [CM94a]: selection of a slotvalue (denoted by an infix operator . here), deletion of a slot (`delete`), and modification of a slotvalue (`update`). Modification works as extension if the named slot is not present in the parameter frame and as deletion followed by an extension otherwise. In [CM94a], extension was chosen as primitive (instead of modification) and was only allowed if the slot to be added was not already present in the frame. In our language, there is no way to statically guarantee the absence of a slot, and each use of such a restricted extension operation would have to be protected by an explicit test for the presence of the named slot. Therefore, we decide to choose modification as primitive here. There are also two primitive operations to test for the presence of a particular slot in a frame (`test`) and to return the list of slotnames for the slots

in a frame (`slots`).[6] The reduction rules are summarized in figure 5.11: frames are represented as indexed sequences of slots, which allows for a concise definition of the operations in terms of index set manipulations. For instance, the value of a selection from a frame (first rule) is the value of the slot that has the selector string s as its name (as usual for primitive operations, selections from frames that do not have such a slot are simply irreducible), and `update` first restricts the sequence of slots to those that do not have the selector string s as their name and then adds a new slot (with the selector string s as its name and the parameter e as its value). More complex operations on frames can be defined in terms of the set of primitives given here.

The basic idea of building a module system in this simple language design is as follows: modules containing function and expression definitions are modeled as records, which contain slots with functions and other expressions as slotvalues, and slotnames are used to represent names of module components as strings. Parameterized modules can be expressed as functions that have modules as results, import relations can be expressed either implicitly, referring to variables bound to modules, or explicitly, passing the imported modules as parameters to the importing modules. Import of components from modules corresponds to selection of values from frames. Using interactions, modules can be stored in files and retrieved from there to be reused in other programs. The explanation is very brief here because more detailed descriptions and a series of examples are provided in chapter 6, which explicitly focuses on techniques for modular programming.

[6] The latter may have unwanted properties if it reflects the order in which slots have been added. It can be made to depend only on the presence of slots if the slotnames are returned in a sorted list.

Chapter 6

Abstractions for Modular Programming

The formal presentation of our language in chapter 5 has demonstrated the simplicity of the design and the influence of the design principles on design decisions. It is the purpose of this chapter to demonstrate, mainly by means of examples, the expressiveness of the resulting language which derives from the generality and orthogonality of the three major parts of the design. In doing so, we provide further evidence that the design goals have been reached, but we also hope to hint at the prospects for languages built according to our proposed design. The chapter concludes with a discussion of the modeling techniques used in the examples which shows how the language design supports these techniques, providing pragmatic a posteriori support for the design principles used.

6.1 Modules

A flood of language constructs has been developed to support modular programming. Chapter 4 lists some of them, and new variants seem to be proposed with every new language. Given that our language includes virtually none of these special purpose constructs, it is not immediately obvious how it can provide support for modular programming. Therefore, we first give some examples of how the effects of some of the special purpose constructs can be achieved. Of course, the main emphasis is on sophisticated module facilities, developed in this section. To begin with, Figure 6.1 shows a typical example program without modules, consisting of several mutually recursive function definitions and a goal expression. In order to focus on program structure, the right-hand sides of function definitions have been omitted. The program defines some operations on binary lists, including constructors (`cons`, `nil`), selectors (`head`, `tail`), a test (`empty`), and a few well-known higher-order functions to `map` a function elementwise over a list, to `filter` out a sub-list of elements, each fulfilling a given predicate, and to `fold`

the elements of a list, using a binary operation. The result of the program is the mapping of the function `square` over a list of three elements.

```
letrec
  empty l    =    ...
  nil        =    ...
  cons h t   =    ...
  head l     =    ...
  tail l     =    ...
  map f l    =    ...
  filter p l =    ...
  fold f c l =    ...
  square x   = ( x * x )
in ( map square ( cons 1 ( cons 2 ( cons 3 nil ) ) ) )
```

Figure 6.1: A simple example program – goal expression view

Even without modules, functional abstraction can be used to reduce the complexity of program design and management. Our example program could also be written directly in terms of the built-in functions and data structures:

```
( < (1 * 1) > ++ ( < (2 * 2) > ++ ( < (3 * 3) > ++ < > ) ) )
```

We have chosen a representation of binary lists in terms of the built-in lists (flat sequences of expressions in angle brackets) here, and have implemented construction of binary lists via the built-in infix operator for concatenation of flat lists (`++`). Other choices would have led to different forms of the program.

With functional abstractions, the example program has been factored into a definition part and a rather simple goal expression (cf. figure 6.1). The program text itself has become more complex in this variant, but the complexity of program maintenance is reduced in several ways: common program parts are abstracted out of the goal expression, and these abstractions are shared – they are instantiated several times (`cons` and `square`). The higher-order function `map` allows to distinguish the operation (`square`) from the way it is applied (to all elements of a list). If we intended to change the representation of binary list, e.g., to nested lists, reversed lists, or search trees, substantial and repeated changes would be required for the direct program version, whereas a redefinition of a few functions would suffice for the variant with abstractions. The relation of the squared list to the original list is not even explicitly expressed in the direct version, and a complete rewrite would be necessary to apply a different function to the list elements, or to apply the same function to a different list.

```
...
map f l  = if ( empty l )
           then nil
           else ( cons ( f ( head l ) ) ( map f ( tail l ) ) )
...
```

Figure 6.2: Example program – one of the function definition views

Apart from these advantages of sharing[1], programming with abstractions also allows several views of the program, each one abstracting away some details of the full program. Suppressing details of the function definitions, figure 6.1 focuses on the goal expression. Only the goal expression, the names of defined functions and an informal understanding of these functions are required in this view, while the right-hand sides of function definitions are not. One of the function definition views is given in figure 6.2. The complete program consists of all views, but functional abstraction allows programmers to concentrate on small parts of the program, substantially reducing the complexity they have to cope with in each phase of the program design. We have used dots here to represent program parts we are not currently interested in, but implementations of the language should also support such partial views of programs, always showing only the parts of programs that correspond to the current focus of attention.

If programs get larger, the program structure needs to be changed because simply adding new function definitions would make the interface to be used between the partial views too complex. The interface would essentially be the complete list of function definitions. Even for this simple example, we have omitted the definition list in the function definition view (figure 6.2), assuming that the interface is obvious, but this is certainly not a valid assumption for larger programs. In the context of the current section, the obvious idea would be to partition programs into modules, where each module would correspond to a partial view of the program and the interfaces between modules would be kept simple. While special purpose constructs such as modules can thus provide guidance for the decomposition of programs, we argue that it is better to start with problem decomposition first and to look for language support later. In the running example, it is possible to introduce an additional level of abstraction to distinguish between the basic functions that define a representation of binary lists and the higher-order functions built on top of the representation functions. For more complex problems, such an approach may lead to a hierarchical decomposition consisting of several levels of abstraction.

To accomplish this in the given language, as a first idea, we may try to use local

[1]Note, however, that this sharing of abstractions would be counterproductive if we would want to treat each list element differently in a program modification.

definitions to hide the low-level details inside the definitions of the higher-level abstractions. However, there is a problem here because low-level functions are usually shared by several high-level functions. In our example, `map`, `filter`, `fold`, and the goal expression all need access to the basic operations on binary lists. On first sight, lexical scoping seems to prohibit the use of locally defined defined functions outside their scope[2], but if the definitions of the low-level operations need to be given repeatedly for each definition of a higher-level operation, the result is obviously inferior to the original program (cf. figure 6.4).

```
letrec
  empty l    =    ...
  nil        =    ...
  cons h t   =    ...
  head l     =    ...
  tail l     =    ...
in letrec
     map f l    = ...
     filter p l = ...
     fold f c l = ...
     square x   = ( x * x )
   in ( map square ( cons 1 ( cons 2 ( cons 3 nil ) ) ) )
```

Figure 6.3: Example program – block structure and sharing

The usual trick used to avoid repetition in functional languages is again based on functional abstraction: the repeated definitions are abstracted out of the whole expression and can then be shared. The result (cf. figure 6.3) is only slightly better than the original program, but programmers can now decide either to look at functions on all levels or to limit their focus of attention to the high-level functions. However, the definitions of low-level functions are not available when programmers focus on the high-level functions: in the goal expression of the outermost `letrec`, the names of the low-level functions occur as free variables – only in the full program are these variables bound. Another problem is that there is no helpful program structure for functions on the same level of abstraction: if we were to introduce additional data types and their operations (trees, arrays, ...), the definitions of these operations would not be separated in any way from the definitions of the basic list operations. As a consequence, the environment for the higher levels of abstraction would be burdened with bindings for all these operations, even though, e.g., the higher order list operations depend only on the basic list operations (and not on the basic tree or array operations). The

[2]As we will see below, this is actually not the case.

```
letrec
  map f l    = letrec
                 empty l    =    ...
                 nil        =    ...
                 cons h t   =    ...
                 head l     =    ...
                 tail l     =    ...
               in ...
  filter p l = letrec
                 empty l    =    ...
                 nil        =    ...
                 cons h t   =    ...
                 head l     =    ...
                 tail l     =    ...
               in ...
  fold f c l = letrec
                 empty l    =    ...
                 nil        =    ...
                 cons h t   =    ...
                 head l     =    ...
                 tail l     =    ...
               in ...
  square x   = ( x * x )
in letrec
     empty l    =    ...
     nil        =    ...
     cons h t   =    ...
     head l     =    ...
     tail l     =    ...
   in ( map square ( cons 1 ( cons 2 ( cons 3 nil ) ) ) )
```

Figure 6.4: Example program – the problems of block structure

higher the level of abstraction in this approach, the larger is the collection of variable bindings programmers have to know about, not to speak of the problems of namespace management (for instance, variants of `map` can be defined for trees and arrays, but a suitable naming convention is necessary to distinguish between these operations).

What has happened to the program structure in the step from figure 6.4 to figure 6.3 is essentially a complete reversal: instead of low-level function definitions hidden inside the high-level ones, the definitions of low-level functions establish the environment in which the high-level functions are defined. The original intention was to introduce a tree-shaped decomposition of programs utilizing block structures to make each block of function definitions (corresponding to a level of abstraction) the root of a new program tree built from lower-level abstractions. However, the scope of definitions extends only towards the leaves of these program trees – lexical scoping does not allow to distribute locally bound identifiers outside their defining expressions. Since local definitions cannot be shared globally, it was necessary to turn them into global definitions and to let the tree grow in the other direction, with low-level definitions at the root and high-level definitions at inner nodes of the program tree. While it seems attractive to build stacks of collections of definitions for small programs, we have argued that it is actually not practical for large programs to have the high-level definitions buried inside all the low-level definitions, or to have only one big collection of definitions for each level of abstraction.

None of the program structures proposed so far supports reuse very well: only individual functions and complete goal expressions can be reused and even they only in the unlikely case that they do not depend on other functions. What would really be necessary are collections of definitions as data objects and abstraction over such collections. This way, the advantages of the structure used in figure 6.3 would carry over to larger and more complex programs, and definitions could be organized in separate structures independent of the level of abstraction they belong to. Unfortunately, the definition parts of `letrec`-constructs do not form valid language expression, so collections of definitions cannot be easily shared with or reused in different goal expressions (other than by copying source text). These definition parts are collections of bindings of identifiers to definitions and provide environments in which expressions with free occurrences of variables can be evaluated. Allowing collections of bindings to be used independently of goal expressions (a kind of *first-class environments*) would immediately conflict with lexical scoping, a language feature much too valuable to give up. To realize the problems, imagine an expression *expr* that is to become the goal expression part of a dynamically substituted environment *env*:

λ $\boxed{env}$. `let`
$\boxed{env}$
`in` *expr*

(the variable *env* is boxed to emphasize that this example is *not* valid in our language) Whether or not variables that occur free in *expr* will be bound in the combined expression depends on the environment substituted for *env* – the binding structure is not obvious until the final expression is actually constructed at runtime, and it depends on the environment substituted for *env*.

This is a serious problem, affecting the very core of our language semantics: binding structure and substitution. It can only be avoided if the binding structure can be determined statically, which means that environments cannot really be first-class objects. Whatever operations are allowed to compute an environment that is to be dynamically attached to an expression, it must be possible to determine statically where (and if) the identifiers that are used in the expression are bound in the environment. At least, restrictions need to be imposed on the permissible operations on environments, similar to the restrictions generally present in statically typed languages. This leads to unfortunate interactions between the design of the static and the dynamic parts of languages that use (first-class) environments for modular programming but do not want to give up lexical scoping, e.g., Standard ML [Mac85] or the language presented in [Jag94].

For these reasons, we refrain from using first-class environments for modular programming. This decision forces us to take a closer look at the restrictions imposed on the use of local definitions by static scoping, and we find that only identifiers cannot leave their local scope. Contrary to the first impression, lexical scoping does *not* prohibit the use of locally defined functions outside the scope of their local identifiers, and indeed, our language allows (anonymous) functions to be used as first-class data objects (a property inherited directly from the λ-calculus). The next steps are straightforward: if collections of lexically scoped bindings cannot be used to support reuse, collections of expressions have to be used instead, i.e., data structures containing functions. Frames are included in the language exactly to provide for such collections, allowing slotvalues to be accessed using slotnames, and (anonymous) functions can be placed in frame slots because they are first-class values. Slotnames are simple string constants, used as selectors in frames, and they are not subject to the restrictions imposed on variable names by static scoping.

Figure 6.5 gives an example of how frames can be used to make collections of functions reusable. With this program, we take up the idea to use local definitions for program structuring again, and we use frames to circumvent the limitations of lexical scoping. This roughly corresponds to the step from the block-structured Algol 60 to the class-structured Simula 67: local values of blocks are made accessible to other program parts. The main differences are that local identifiers are not made accessible, only their values, and that there is no update operation for identifiers.

Basically, the function definitions are grouped into the basic list representation (`basic_list`) and the higher-order functions on lists (`hof_list`), and frames are used to make (some of) the locally defined functions accessible from

```
letrec
  basic_list = letrec
                 empty l  =  ...
                 nil      =  ...
                 cons h t =  ...
                 head l   =  ...
                 tail l   =  ...
               in { "empty" :: empty
                    "head"  :: head
                    "tail"  :: tail
                    "nil"   :: nil
                    "cons"  :: cons }
  hof_list repr = letrec
                    empty      = ( repr . "empty" )
                    head       = ( repr . "head" )
                    tail       = ( repr . "tail" )
                    cons       = ( repr . "cons" )
                    nil        = ( repr . "nil" )
                    map f l    =  ...
                    filter p l =  ...
                    fold f c l =  ...
                  in { "map"    :: map
                       "filter" :: filter
                       "fold"   :: fold }
in letrec
     hof      = ( hof_list basic_list )
     cons     = ( basic_list . "cons" )
     nil      = ( basic_list . "nil" )
     map      = ( hof . "map" )
     square x = ( x * x )
   in ( map square ( cons 1 ( cons 2 ( cons 3 nil ) ) ) )
```

Figure 6.5: Modular version of the example program

outside, i.e., frames are used similar to *export interfaces* of modules. The module `basic_list` consists only of local definitions and an export frame that makes all local functions accessible. The module `hof_list` is a little bit more complex in that its function definitions depend on the functions defined in `basic_list`. In conventional module systems, such a dependency would be hard-coded using an *import declaration* in `hof_list`. Similarly, the definition of `hof_list` could refer directly to `basic_list` in our language, but we prefer to abstract from the representation module on which `hof_list` depends, making it a parameter of the module definition. Inside the definition, the functions that are needed from the representation module can be used via frame selection from the parameter, which would be similar to the use of *qualified names* in conventional systems. Additionally, abstraction is used here to share the selections, binding them to local variables which corresponds to the specification of an *import interface*. In the goal expression of the top-level `letrec`, `hof_list` is applied to `basic_list` to yield a module `hof` of higher-order functions working on this particular representation of binary lists. Finally, the functions `cons` and `nil` from `basic_list` and `map` from `hof` are imported and the rest of the program is as in figure 6.1.

The definition of `hof_list` in figure 6.5 demonstrates the main features usually attributed to modules: it has a collection of local definitions and explicit import- and export-interfaces (the selections from the parameter `repr` and the frame that is the value of `hoflist`). Furthermore, it is parameterized, and the parameter is itself a module, by which fact `hof_list` can also be seen as a simple function having modules as parameter and result. Because of the close correspondence of frames used in this way to explicit module constructs in conventional languages, we refer to these frames as *modules*. The construction of these frames may depend on local definitions (which are not externally visible) and on the values substituted for variables that occur free in the frames. If the free variable occurrences are bound by the formal parameters of a function that returns such a frame as its value, we also refer to the function as a *parameterized module*. Due to the use of modules as parameters, parameterized modules such as `hof_list` can be reused with different imports, providing a flexibility not present in conventional module systems but available, e.g., with Standard ML's functors. However, the current definition of Standard ML's module system (for the issues discussed here, [Mac85] is still accurate) does not allow to pass functors as parameters, making it necessary to resolve all imports of a parameterized module in the program's top-level definitions before such a module can be passed as parameter to other modules. Some ML implementations, e.g., Standard ML of New Jersey [CM94b] go beyond this and allow to pass parameterized modules as parameters as well, providing for *higher-order functors*. Of course, this is possible in our language, too, and we can do even more because modules and module parameters are dynamic, first-class values in our language. In contrast, Standard ML has a stratified design, where modules are compile-time values different from runtime values. For instance, modules can depend on runtime values or can be

selected using conditional expressions in our language, but not in Standard ML. The rationale of the decision for the stratified design of Standard ML is that it is a statically typed language, and that its modules can contain type information [Mac85].

Note that a hierarchical problem decomposition need not lead to hierarchical programs: although `hof_list` depends on a representation module, the modules corresponding to both abstraction levels are treated equally and may be reused independent of each other. Note also that associations between local identifiers and slotnames have to be given explicitly both for import and export: name clashes due to implicit import of items exported with the same name from several modules cannot occur, qualified names and renaming of imported items fall out naturally in the frame model. Still, it would be nice to have some syntactic sugar for trivial import- and export-interfaces. Two examples are given in figure 6.6: a collection of definitions inside frame braces could be converted into a collection of definitions followed by an export frame, an import declaration in the definition part of a `letrec` could be converted into a list of definitions involving frame selections. Using this sugar, the program in figure 6.5 can be rewritten as in figure 6.7. The program is still a bit longer than the original version of figure 6.1, but this is to be expected since modular structure has been added to the program.

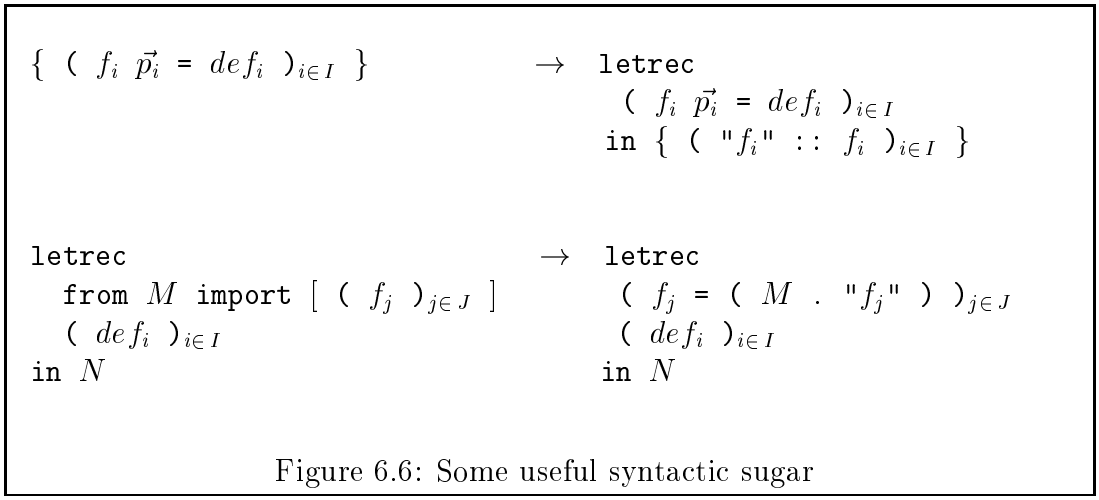

Figure 6.6: Some useful syntactic sugar

One may be tempted to ask whether all the flexibility of first-class modules is really needed. Therefore, it seems advisable to point out that we have **not** described a complex module language that would have to be defined and implemented on top of the programming language – on the contrary, there is **no explicit module language** in our design. Instead, the programming language is expressive enough to provide for all the flexibility usually associated with quite complex module languages. The main reason for having the potential of first-class modules in our design is that we do not have explicit module constructs at all –

```
letrec
  basic_list = { empty = ...
                 nil   =  ...
                 cons  =  ...
                 head  =  ...
                 tail  =  ...  }
  hof_list repr = letrec
                    from repr import [ empty head tail cons nil ]
                  in { map     =  ...
                       filter  =  ...
                       fold    =  ... }
in letrec
     from basic_list import [ cons nil ]
     from ( hof_list basic_list ) import [ map ]
     square x = ( x * x )
   in ( map square ( cons 1 ( cons 2 ( cons 3 nil ) ) ) )
```

Figure 6.7: Modular version of the example program, with syntactic sugar

we just make use of the data structuring facilities of our programming language. This is the real meaning of the slogan: *modules should be first-class data structures*. Asking for any restrictions as far as modular programming is concerned would thus be asking for restrictions of the programming language itself.

Note that we have used essentially the same program for the whole discussion. The program variants presented so far do not differ in the result of the given goal expression, they only differ in their program structures. Although each variant would serve the purpose of computing one particular program result, the variants differ widely in how they factor the whole program into separately understandable parts, how well they support reuse of these program parts, and how complex the interfaces between the parts are. Programs composed of several parts correspond naturally to problem decompositions and help to reduce the apparent complexity of programs. The overall complexity of the complete program text is not reduced (apart from the effects of sharing) – programs may even get more complex due to the additional program structure, but this additional structure allows to understand the complete program in terms of several partial views, each one simple enough to be easily and separately understood. Similarly, defining the program parts in such a way that they may be reused in other programs complicates the initial task of just producing a working program, but pays off in the long term. Finally, the complexity of the interfaces between program parts is not only important for the potential reuse of parts, but also for program maintenance and evolution: complex interactions between the parts make program modifica-

tions difficult, whereas uncompromising separation and simple interfaces render it more likely that consequences of changes can be localized to a few parts.

Therefore, modular programming is first of all associated with additional efforts. If programs are very large or complex, or subject to frequent changes, the advantages following from the additional efforts may manifest themselves in a single project. More commonly, however, the virtues of modular programming become apparent only if modules can be reused in several programs, which raises the issue of how this can be achieved in our language. In conventional languages, module storage and import happen at the borderline between programming language and operating system and, even if programmers need not descend to the level of operating system tools to compose their programs, the semantics of import declarations does not follow the usual language semantics. In contrast, our language design allows us to move from these ad-hoc approaches to a more language-conforming way of handling modules, and it is our goal to extend this high-level approach to the issues of long-term module storage. Note again that modules in our terminology are not source code structures but first-class data structures.

Basically, modules are constructed dynamically in the evaluation of programs. In order to make these modules available to other programs, it is necessary to establish a communication between programs, a communication involving modules. Additionally, there need not be any time interval in which both producers and users of modules are active – the communication needs to be buffered in a long-term storage that serves as an environment for program development. Described in this way, this is exactly the kind of problem for which we have introduced interactions into our language design: communication of programs with a runtime environment. Since we have not restricted the input/output-facilities of our language to sequences of characters, we can use the file system as a long-term storage for modules. Because functions and modules containing functions are first-class values, and every valid language expression can be the object of an interaction, formally described language features (interactions) can be employed to handle the storage of modules in files and the import of modules from the file system. Languages in which functions or modules are not first-class values with respect to input/output need to be extended with special purpose constructs to accomplish this. Users of these languages often suffer from the need to leave the language level for program construction, using operating system tools for this essential phase of program development. Only if it is no longer necessary to refer explicitly to features outside the language definition, the whole business of program construction can be lifted to the language level, hiding any implementation details such as compilation and linking below this level (just as the details of memory management are left to implementations).

In our running example, the modules `basic_list` and `hof_list` can be packaged together to form a `list_library`, which is just a collection of modules, and stored in the environment, using a file with this name (figure 6.9). To avoid clut-

```
onFile file prg = ( ( fopen file ) >>= λfile_handle.
                  ( ( prg file_handle ) >>= λresult.
                  ( ( fclose file_handle ) >>
                  ( return result ) ) ) )
load file       = onFile file fget
store file expr = onFile file λfile_handle. fput file_handle expr
```

Figure 6.8: Abstracting away some file handling

```
( store "list_library"
  { basic = { empty = ...
              head  = ...
              tail  = ...
              nil   = ...
              cons  = ... }
    hof   = λrepr. letrec
                     from repr import [ empty head tail cons nil ]
                   in { map    = ...
                        filter = ...
                        fold   = ... } } )
```

Figure 6.9: Creating a persistent library

```
( ( load "list_library" ) >>= λlist_library.
    letrec
      basic    = ( list_library . "basic" )
      hof      = ( ( list_library . "hof" ) basic )
      map      = ( hof . "map" )
      cons     = ( basic . "cons" )
      nil      = ( basic . "nil" )
      square x = ( x * x )
    in ( map square ( cons 1 ( cons 2 ( cons 3 nil ) ) ) ) )
```

Figure 6.10: Using a persistent library

tering programs with low-level file handling, a function `onFile` is used to define `load` and `store` (figure 6.8). `onFile` takes a filename and a program, opens the named file and passes the file-handle as a parameter to the program. After the program has produced a result, the file-handle is closed and the program's result is returned. `load` and `store` use `onFile` to get one expression from a file and to put one there, respectively. These and other similar abstractions to raise the level of interactions are collected in a library of input/output-operations (a default treatment of non-executable interactions is also included in the definitions of these library functions but is not shown here). Afterwards, any other program can refer to the file `list_library` to use the library. It just loads the library from the environment and extracts the modules `basic` and `hof`, instantiates `hof` to use the representation defined in `basic` and proceeds as if the modules had been defined in the same program (figure 6.10). Similar to slotnames, filenames have to be explicitly associated with local variables, and it may be useful to introduce syntactic sugar for the two standard forms of `store` and `load`. In this case, it may even be necessary to include the code for `onFile` in the syntax extension to avoid a bootstrapping problem: to use a function from the input/output-library, the library needs to be accessed, which is exactly the task for which the library function should be used. Nevertheless, such a language extension is only a matter of convenience and its semantics can be defined in terms of the available primitives. Therefore, our goal to lift the treatment of module storage, module load and program construction to the language level has been achieved.

6.2 Data abstraction and generic functions

In the early 1970s, it was noted (cf. chapter 4) that modules are often used to define abstract data types. In our example, the collection of functions in `basic_list` can be interpreted as a definition of the type binary lists: in Russell terms, the functions interpret values of some value space as binary lists. Usually, the kind of values that are actually used in such an interpretation is restricted and can be viewed as a hidden type. Taken together, a value of a hidden type and a collection of functions providing an interpretation of this hidden type as a new (user-defined) type form a representation of an *abstract data type*. In general, there are many possible representations of an abstract data type, each providing the same operations on the abstract data type but using different implementations or different hidden types. This different interpretation of collections of functions influences the structure of our example (repeated in figure 6.11).

`basic_list` has been renamed to `binary_list` to emphasize the change of intent: it is no longer a module of some basic operations on lists, it is a definition of binary lists in terms of one possible representation. Similarly, `hof_list` is a parameterized definition of lists with some higher-order functions, defined in terms of a representation of binary lists. Obviously, this program is a gross

```
letrec
  binary_list = { empty = ...
                  nil   =  ...
                  cons  =  ...
                  head  =  ...
                  tail  =  ...  }
  hof_list repr = letrec
                    from repr import [ empty head tail cons nil ]
                  in { map     =  ...
                       filter  =  ...
                       fold    =  ... }
in letrec
     from binary_list import [ cons nil ]
     from ( hof_list binary_list ) import [ map ]
     square x = ( x * x )
   in ( map square ( cons 1 ( cons 2 ( cons 3 nil ) ) ) )
```

Figure 6.11: Modules as simple data abstractions?

over-simplification of abstract data types as no type checking is done. Instead, operations of the 'right' type are explicitly selected and only applied to objects belonging to the same representation of this type, e.g., `(binary_list . "cons" )` is applied to `(binary_list . "nil" )`. Moreover, the generic 'type' `hof_list` provides no constructors, making it impossible to construct objects of the abstract data type `(hof_list repr)` without knowledge of the 'hidden' representation type `repr`. Finally, the function `map` belonging to the abstract data type `(hof_list binary_list)` is applied to an object of the 'hidden' representation type `binary_list` outside the definition of the abstract type. To sum it up, this naive attempt shows some similarities to abstract data types, but it is also in conflict with the philosophy of abstract data types. Thus, a change in program structure is necessary to reflect the change of intent.

Since we have excluded aspects of type systems from our language design, we cannot fully address the type safety aspects of abstract data types here. There is no static type checking, there are no means to query the type of an expression at runtime, we have not even defined a type system. For these reasons, we prefer to talk about data abstractions instead of abstract data types here. Note, however, that type systems that do not provide for abstract data types at all or cannot relate their concept of abstract data types to our data abstractions would not be acceptable for our language. With these restrictions in mind, the program structure still has to be changed to reflect the idea of data abstractions. First of all, each abstract value has to be represented by a package, consisting of a value

of a representation type[3] and a collection of functions implementing the abstract operations on the representation type. Taken together, the representation type and the collection of functions defined on it form one possible representation of an abstract value, and frames with two slots named `"type"` and `"value"` can be used to describe such a representation.

```
letrec
  binary_list = { empty = ...
                  nil   =  ...
                  cons  =  ...
                  head  =  ...
                  tail  =  ...  }
in letrec
     wrap v t = { "type"  :: t
                  "value" :: v }
     nil      = wrap ( binary_list . "nil" ) binary_list
     cons h t = wrap (( binary_list . "cons" ) h ( t . "value" ))
                     binary_list
   in ( cons 1 ( cons 2 ( cons 3 nil ) ) )
```

Figure 6.12: A simple data abstraction

Figure 6.12 shows a simple attempt to use the collection `binary_list` in the definition of a data abstraction: the operations from `binary_list` and the concrete values on which the definitions of these operations are based are lifted to abstract values by wrapping them into appropriate packages. Correspondingly, abstract values may have to be unwrapped before operations from the representation can be applied (cf. the definition of the abstract `cons`). In this attempt, the abstract constructors `nil` and `cons` are defined in terms of implementations of these constructors in a single representation (`binary_list`). It would be annoying to modify the definitions for every new representation, and it would also lead to naming conflicts if multiple representations of the same data abstraction are being used in one program. The problem is aggravated by the fact that multiple data abstractions may provide similar functions using the same name (e.g., `map` is a typical example of a function that will be provided for most kinds of data collections). A common solution to this kind of problems is the introduction of *generic functions* or *overloading*. A generic function may behave differently depending on the type of its parameters, i.e., the name of a generic function is overloaded with the definitions of all data abstractions (or representations thereof) that support a function of this name.

[3] in the sense of a syntactical category or another abstract data type

```
( store "generics"
 { genConst c ty   = ( ty . c )
   genOp op da     = (( ( da . "type" ) . op ) ( da . "value" ))
   genOp2 op x da  = (( ( da . "type" ) . op ) x ( da . "value" ))
   wrapConst gc ty = { "type"  :: ty
                       "value" :: ( gc ty ) }
   wrapOp op da    = { "type"  :: ( da . "type" )
                       "value" :: ( op da ) }
   wrapOp2 op x da = { "type"  :: ( da . "type" )
                       "value" :: ( op x da ) } } )
```

Figure 6.13: Helper functions for the construction of generic definitions

Figure 6.13 shows a module of definitions from which generic functions can be constructed. For simplicity, we assume that the functions are generic in only one of their two first parameters (which is enough for our running example). The essence of a generic function is given in `genOp`. The parameter to a generic function `( genOp op )` is an abstract value: from its `"type"`-slot, the actual type-specific function definition is selected and then applied to the concrete value in the `"value"`-slot of the abstract value. `wrapOp` is used to wrap the concrete result of the concrete function into an abstract value (`genOp2` and `wrapOp2` do the same for the case that the second parameter of the generic function is the abstract value; the first parameter `x` is passed on unmodified in this case). The only difference for overloaded constants (`genConst`, `wrapConst`) is that there is no abstract value from which the type can be selected – the type has to be passed as an explicit parameter.

Figure 6.14 shows how generic functions are constructed from the definitions in `"generics"`. Note that the definitions of the generic functions `cons` and `map` and of the overloaded constant `nil` are independent of any specific type and apply equally well to all data abstractions that supply similar functions and constants. For instance, the definition of `hof_list` has been modified to extend the representation supplied as parameter, i.e., it exports not only the higher-order functions, but also the representation functions. Since the re-exported items are not modified, `(hof_list binary_list)` is a kind of sub-type to `binary_list`, and all abstract values of the former type can be used where an abstract value of the latter type is expected. In particular, generic functions that apply to `binary_list` can also be used with abstract values of its sub-type. The overloaded constant `nil` is instantiated to `hof_nil`, which is comparable to a type annotation. All other generic operations select type information from their parameters, so that types are handled implicitly, and the final goal expression remains almost unchanged.

```
(( load "generics" ) >>= λgenerics.
 letrec
   from generics import [ genConst wrapConst genOp2 wrapOp2 ]
   binary_list = { nil   =  ...
                   cons  =  ...
                   empty =  ...
                   head  =  ...
                   tail  =  ... }
   hof_list repr = letrec
                     from repr import [ empty head tail cons nil ]
                   in { nil     = nil
                        cons    =  cons
                        empty   =  empty
                        head    =  head
                        tail    =  tail
                        map     =  ...
                        filter  =  ...
                        fold    =  ... }
 in letrec
      nil      = ( wrapConst ( genConst "nil" ) )
      cons     = ( wrapOp2 ( genOp2 "cons" ) )
      map      = ( wrapOp2 ( genOp2 "map" ) )
      hof_nil  = ( nil ( hof_list binary_list ) )
      square x = ( x * x )
    in ( map square ( cons 1 ( cons 2 ( cons 3 hof_nil ) ) ) ) )
```

Figure 6.14: Using data abstractions for the example problem

Usually, data abstractions that provide implementations of generic operations are somehow related and one could try to organize data abstractions in a hierarchy just as the representations of data abstractions form a two-level hierarchy in our example. In other words, collections of generic operations could be built similar to collections of representations as this is done, e.g., with type classes in Haskell (type classes correspond to collections of generic operations which are organized hierarchically and instances of type classes correspond to collections of concrete function definitions that characterize a type). Of course, the flow of type information is restricted in our model, compared with the type inference process in Haskell where, e.g., the type of identifiers overloaded with constants of different types can often be inferred from the context). Also, the information gathered during type inference can be used to modify program representations prior to execution to supply type representations as parameters implicitly without bothering programmers with the details of data abstraction (such as wrapping, unwrapping, selection of functions from type representations, etc.). Thus it seems that type systems provide for a useful form of meta-level abstraction that is not yet easily supported in our language.

6.3 Object-oriented programming

In the series of program variants developed so far, the emphasis has shifted more and more from functions towards data structures, and the program structures reflect this shift of emphasis. In the original version of our running example (6.1), definitions of algorithms were kept with functions, and values were only supplied as parameters to these functions. Then, functions were collected into modules (6.5), collections of functions were interpreted as definitions of types (6.12), and functions and values were packaged to form data abstractions (6.14). Based on these data abstractions, generic functions have been introduced in the last section, functions whose behavior depends on the type of their parameters and whose definitions are supplied by the representations of these types. In our final example, we arrive at the opposite end of the spectrum, where definitions of algorithms are kept with the objects, and functions (degenerated to messages) are only supplied as parameters to these objects. Our running example originated from a function-oriented view of programs where it is natural to think about lists and higher-order functions on lists (indeed, they are typically pre-defined in functional languages). This may not at all be a typical application from an object-oriented point of view, i.e., not only the program structures are different, but problems present themselves not usually in terms of lists and functions, and the example should therefore only be seen as a simple representative of more complex programs. However, before we can proceed to restructure the example, a framework for object-oriented programming has to be defined similar to the framework for generic functions in the previous section.

```
find_method class message =
  if ( test class message )
  then ( class . message )
  else if ( test class "super" )
       then ( find_method ( class . "super" ) message )
       else "message not understood"
send oid message things =
  if ( test things oid )
  then let
         class = ( ( things . oid ) . "class" )
       in ( find_method class message )
  else "object not found"
```

Figure 6.15: Finding methods in the class hierarchy in response to messages

If programs are to be organized around objects and algorithm definitions are to be stored in objects, there must also be a protocol for accessing and invoking these algorithms. In Smalltalk terminology [Ing78], the metaphor of communicating objects is used for this purpose: algorithms defined in objects are called methods and are invoked by sending messages to objects. To be precise, objects are instances of (object) classes, and the methods to which an object responds are defined in its class or in one of the super-classes of this class (classes are organized in a hierarchy). Obviously, frames can be used as representations of objects and classes if these are viewed as collections of object components (instance variables) and method definitions, respectively. Similarly, collections of objects and hierarchies of classes can be represented as frames, provided that a suitable message sending operation can be defined. The basic algorithm is given in figure 6.15: given an object identifier, a message, and a collection of objects, `send` determines whether the object addressed by `oid` does exist, and calls `find_method` with the object's `class` and the `message` if it does. `find_method` selects the appropriate method if it exists in the class and recurses upwards (along the `"super"`-slots) in the class hierarchy otherwise. If the top of the class hierarchy is reached before a method is found (no `"super"`-slot), a message to this effect is returned.

Strictly speaking, object-oriented programming is just a way to organize programs around their data objects. In practice, however, the ability to modify the value of an object without changing its identity is often an essential ingredient of object-oriented programs. The reason for this is that the collection of all objects is used as a communication medium: to make an information available to other objects, it is stored in one of the publicly known objects. These objects are known not by their values, but by object identifiers, and if an object is modified, its new value is available under the old identifier. To model this common feature,

the collection of all objects is also represented by a frame, and slotnames can be used as object identifiers (the collection of objects is called `things` here). Furthermore, a monad is used to organize accesses to the collection, which means that all methods have to be constructed in this monad. Figure 6.16 shows the slightly adapted send operation in the context of the other operations which we describe now.

The functions `ret` and `bind` are the familiar state transformer monad operations (built on the input/output-monad here to provide for input/output in object methods). All functions in this module take a collection of objects (`things`) as the final parameter and construct an input/output-interaction script that returns a pair consisting of a value and a collection of objects. `ret` takes a value and the `things` and returns both in a pair (mainly used to return results from subcomputations). `bind` takes two scripts in the objects-monad, executes both in sequence and passes the result of the first to the second. `io` lifts a script in the input/output-monad to a script in the objects-monad. Both `bind` and `io` check the intermediate results and pass them on only if they are `OK` (otherwise, the evaluation returns immediately with a message indicating that some operation cannot be executed)[4]. These were only the basic operations to construct programs operating on a collection of objects and performing input/output, whereas the remaining operations define one possible basis for an object-oriented programming style: `new` creates a new object of a `class` where each class is assumed to provide a prototype for such objects in a slot `"new"`. For simplicity, prototypes cannot be parameterized here, and the identity of the new object is provided as a parameter to new (this would better be handled internally by computing a new name for each new object; a name generator could be part of the objects collection). `set` allows to modify components of objects without changing their identity which involves two updates: the object has to be modified, and the collection of objects has to be updated to contain the modified object. Finally, there is the definition of message sending with some small modifications. `find_method` is now local to `send`, and both definitions have been adapted to fit into the objects-monad. More interestingly, methods selected from the class hierarchy take the object that originally received the message as a parameter `self`, and all methods take one additional parameter. Both parameters are supplied to the result of the call `(find_method class message)` in the goal expression of `send`.

Given this library, it is now possible to define the object classes for our running example (figure 6.17). Since our definition of `send` assumes a uniform method interface, each method takes two parameters. Most of them do only need the `self` parameter to know the object to which a message is addressed, while `fold` actually needs more than two parameters. `binary_list` is now an object class, and new lists are initially empty, so there is no explicit `nil` method. `hof_list` is

[4]We have silently extended `check` to handle both unary and binary `OK`-constructs, but do not include `things` in `NE`-constructs here.

```
( store "objects"
 { ret v    things = ( return ( OK v things ) )
   bind a b things = ( ( a things ) >>= ( check b NE ) )
   io i     things = ( i >>= ( check λr.( return (OK r things) )
                                    NE ) )
   new class oid things =
     ( return ( OK oid ( update things
                                oid
                                ( update ( class . "new" )
                                         "class"
                                         class ) ) ) )
   set oid selector value things =
     if ( test things oid )
     then ( return ( OK value
                         ( update things
                                  oid
                                  ( update ( things . oid )
                                           selector
                                           value ) ) ) )
     else ( return ( NE "object not found" ) )
   send oid message par things =
     letrec
       find_method class message =
         if ( test class message )
         then ( class . message )
         else if ( test class "super" )
              then ( find_method ( class . "super" ) message )
              else λself.( return (NE "message not understood") )
     in if ( test things oid )
        then letrec
               obj   = ( things . oid )
               class = ( obj . "class" )
             in ( ( find_method class message ) obj arg things )
        else ( return ( NE "object not found" ) ) } )
```

Figure 6.16: An objects monad for message send and object modification

```
( ( load "objects" ) >>= λobjects.
  letrec
    from objects import [ ret bind io new set send ]
    binary_list = { new            = { "val" :: < > }
                    empty self arg = ...
                    cons self arg  = ...
                    head self arg  = ...
                    tail self arg  = ... }
    hof_list = { new             = { "val" :: < > }
                 super           = binary_list
                 map self arg    = ...
                 filter self arg = ...
                 fold self args  = ... }
    things = { }
    square x = ( x * x )
  in ( bind ( new hof_list "a_list" ) λoid.
     ( bind ( send oid "cons" 3 ) λv.
     ( bind ( set oid "val" v ) λv.
     ( bind ( send oid "cons" 2 ) λv.
     ( bind ( set oid "val" v ) λv.
     ( bind ( send oid "cons" 1 ) λv.
     ( bind ( set oid "val" v ) λv.
     ( bind ( send oid "map" square ) ) ) ) ) ) ) ) ) things ) )
```

Figure 6.17: Using the objects monad for the list example

defined to have `binary_list` as its `super`-class and inherits all methods defined there. The goal expression starts with an empty collection of objects and proceeds as follows: first, a new empty `hof_list` object is created and its identifier is bound to the variable `oid`. The object with the identifier `oid` is then asked to return a copy of itself with the value `3` prepended and is `set` to this new value. This is repeated for `2` and `1` and, finally, `oid` is asked to `map` the function `square` over its elements. The program is a bit complicated because our definition of `set` (roughly corresponding to an assignment in an imperative object-oriented language) does not allow any access to `things` on its right-hand side (the third parameter).

We should point out again that this is only a simplified attempt to provide for object-oriented programming, and that there are various other possible approaches to model objects. To give but one example, our definitions assume the classes and the class hierarchy to be fixed, only objects and the collection of objects can be modified. Another approach would be to include the frames that model classes in the collection of modifiable objects. In such an approach, the distinction between objects and classes would vanish – instead of classes referring to super-classes, there would only be objects that would handle some messages themselves but could also delegate messages to other objects. Similarly, variations in the definitions of the other constructs (modules, data abstractions, generic functions) are possible, too, but in general, we have successfully avoided the hammer-and-nail problem[5] at the level of modular programming (if you only have modules/objects/..., every problem looks like a module/object/...). Instead, our language design invites programmers to think about the abstractions they need and to build or use the best tools for a given problem. Useful abstractions can be provided as libraries, but they do not need to be built into the language. Indeed, they should not be built-in, since none of the constructs is really fundamental – all of them can be composed from a few fundamental language constructs. Most of the necessary constructs are available in our design, with the notable exception of the kind of meta-level abstraction provided by type systems. Of course, the problem avoided at the module level returns at the next level because we map all abstractions to λ-abstractions and all collections to frames. We have shown how these tools are adequate for the problems we are interested in here, but we do not claim them to be adequate for all kinds of abstraction or for all kinds of collections.

6.4 Discussion

Although our language design is rather simple, we have been able to model special purpose constructs of more complex languages with relative ease. So far, this has been demonstrated only by a number of examples, but the ideas and

[5] 'If all you have is a hammer, everything looks like a nail.'

modeling techniques used have been rather general, and there is no reason to assume that these techniques would be restricted to the examples given here. This brings up the issue of formalizing these techniques in order to achieve a better understanding of their prospects and foundations.

We believe that the basis for these techniques is a process of abstraction that can be formalized in a way that corresponds to the intuitive use of the term. As a consequence, the limits of these techniques are given by the limits of support for abstraction in programming languages. Moreover, special purpose constructs for modular programming are nothing but instances of abstraction, and the incorporation of such constructs in the definition of programming languages is made necessary only if these languages are restricted in their support for general abstractions.

Throughout this thesis, we have emphasised the importance of abstraction as the major tool for dealing with complexity, and we have done so in the usual ad hoc manner. We generalized concepts by abstracting away details that were not relevant to the issues discussed (the use of the term abstraction for this process in programming seems to be due to Hoare and Dahl [DDH72, III. Hierarchical Program Structures, p. 209]). We will now try to explain the success of our approach by a closer investigation of the process of abstraction, how it relates to the problems we are trying to solve and to the facilities we provide in our language.

The mechanics of abstraction

The idea of abstraction is to factor a problem into two parts in order to focus on one part and to abstract away the other. Figure 6.18 is a first attempt to isolate the major steps involved in the process of forming an abstraction, starting with an object X. First, X is factored into two parts A and B, where $||$ denotes a binary composition operator that constructs the original object from the two parts. Often, one of the parts is a larger context in which the other part is embedded as a sub-object, but using a binary operator to denote the composition of the whole object from the two parts allows to keep the presentation symmetric. The second step is a quantification over the irrelevant part, e.g., for all a, only the composition $a \,||\, B$ may be of interest, whereas the actual value of a is considered uninteresting for the current task. Usually, the uninteresting part is not completely irrelevant and the quantification thus needs to be restricted to a subset of objects ($a \in \mathcal{A}$). Such restrictions have an important influence on the final step: it is usually not possible to address the interesting part of the problem in isolation, ignoring the other part and the composition completely. Instead, at least the restrictions need to be kept with the interesting part, providing an *interface specification*. Only if there are no restrictions, the interface becomes trivial and it is possible to isolate one part. Note that there are always two views of an abstraction. Even though only one of the views may be relevant for a given sub-task, there will always

be a complementary task (however trivial) for which exactly the other view will be relevant. In other words, abstraction provides two simpler, but partial views of the original object X, but it does not usually change the complexity of X. However, the generalization step allows to share partial views between different abstractions, providing a potential for reuse that can actually reduce the overall complexity of the partial views: each partial view can occur at several instances in X.

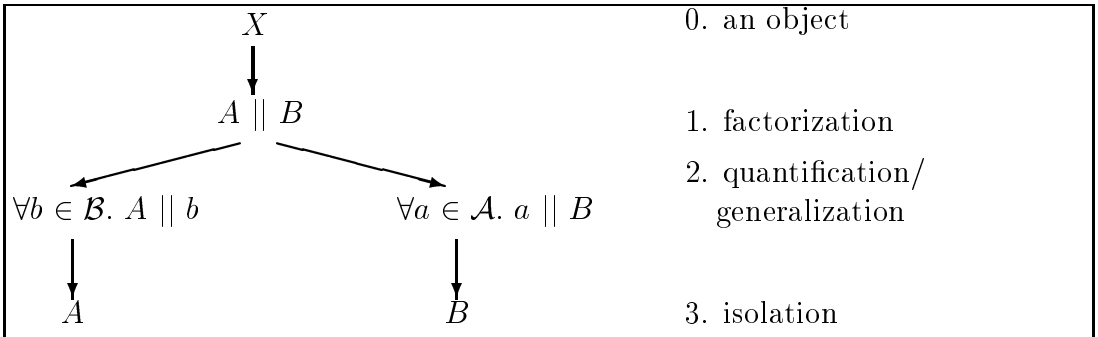

Figure 6.18: The process of abstraction

The objects in our language are expressions, and the obvious representation of $\forall$-quantified expressions are λ-abstractions. Applications, β-reduction and, finally, substitution model the construction of expressions from abstractions and sub-expressions. The other language constructs provide the means for the composition of programs from parts inside abstractions. We are aware that the choices made here are not the only possibilities – they just reflect the tools we used. These may be neither complete nor fundamental (e.g., only valid expressions can be abstracted away), but they seem to be sufficient for our purposes - sufficiently expressive and sufficiently simple. Note, however, the importance of our language design principles for this representation of abstraction: (a) we need to be able to build abstractions over every valid language expression, and (b) we need to be able to abstract away every valid language expression out of any context. Any deviation from the principles of abstraction or data type completeness would hamper our freedom to build abstractions. Such restrictions of the general facilities for abstraction are the major reason for the need to introduce special purpose abstractions as predefined language constructs (e.g., procedures, modules, ...).

We are now in a position to relate the process of abstraction as outlined above to our problems and solutions. As a simple example, consider the application of a function to an argument: functions are just abstractions over expressions, which are factored into varying sub-expressions (parameters) and fixed contexts that describe algorithms in terms of these sub-expressions. There are two quantifications involved: at the call site, the algorithm is replaced by a function name and, at the function definition site, the parameters are replaced by names. As a result, we get two simplified parts of the original program: at the call site, we abstract

away the implementation of the algorithm and, in the function definition, we do not need to care about possible calling contexts but can focus our attention on the algorithm instead. Strictly speaking, the calling context includes not only the actual parameters but the complete expression context in which the function call is embedded. However, the function definitions are independent of these contexts (the reduction of function calls to function values is context-free), and these context can therefore be abstracted away, isolating the function calls from their embedding contexts. In contrast, the separation between function call site and function definition has to stop after the quantification steps: the result of the function call depends both on the actual algorithm and the actual parameters, and therefore the names for the abstracted parts remain in the *interfaces* between the two parts (they cannot be isolated from each other).

If interaction descriptions are added to the valid expressions, the same scheme that is used to build functional abstractions can be used to build procedural abstractions, and if records are added to model modules, module abstractions can be built in the sense of conventional module systems, i.e., module definitions can be understood and maintained while abstracting away the complex programs in which they may be used, and the programs can be understood as a collection of modules abstracting away the complex module definitions. The addition of records allows to model the collection aspect of modules, i.e., modules group together collections of expressions (including functions). In all the examples provided in this chapter that involve the modeling of special purpose constructs such as modules, data abstractions or objects, the basic idea is to identify the abstraction behind the language construct and to implement this abstraction directly without using the special constructs (though the technical details may be complex enough in some of the examples to hide this central idea).

The influence of abstraction on the language design

Since data structures are used to model modules in our language, we get first-class modules for free. But even if we had decided to add modules as separate language constructs, our emphasis on abstraction would have led us to first-class modules again (the same reasoning applies to the other constructs). Without modules being first-class expressions, the means for abstraction would be seriously restricted, e.g., it would not be possible to abstract away common sub-expressions from the collections of expressions in module definitions (parameterized modules) or to abstract away collections of sub-expressions from any valid expression (modules as parameters). As an immediate consequence of the latter restriction, modules that depend on other modules (via an import relation) could not be used with varying imports. In contrast, unrestricted abstraction not only allows to use one module in several programs requiring similar functionality, it also allows one program to use several modules providing similar functionality. This latter quality of multiple implementations for given interfaces is usually attributed to object-

oriented languages and has, e.g., been described as the essential new capability added in the transition from the abstraction-oriented (in the sense of providing for data abstractions) Ada 83 to the object-oriented Ada 95 [Taf93].

The λ-calculus was invented to describe functions and, due to the fact that instantiation of λ-abstractions corresponds so nicely to function application, it has mainly been used as the basis of functional programming languages. However, we prefer to see it as a (probably partial) solution to the problem of describing abstractions. In particular, both functional and object-oriented programming can be supported in a suitably extended λ-calculus – even though functions and objects may not go easily together, they both are just useful instances of the general scheme of abstraction. Whether the abstractions of the λ-calculus model functions, procedures, modules, data abstractions, objects, classes, or something else depends mainly on the available primitives. Consequently, we have tried to keep the pure λ-calculus as the core of our language and have added primitives to address various problem domains: constants, primitive functions, data constructors and selectors, primitive interactions, and so on. In view of this, our language design is better described as consisting of four parts: functions, frames, interactions, and λ-calculus as a means for building abstractions over the available primitives. Also, we have explicitly added records as data structures instead of modifying the λ-calculus to provide for functions with named parameters (see [Dam97] for a nice instance of this approach). This alternative would allow to model records similar to the modeling of other data structures in the pure calculus (as discussed briefly in section 2.3), but it would also blur the distinction between general means for abstraction and specific constructs for data structuring.

Our view of abstraction treats both modules and levels of abstraction as instances of one scheme (perhaps focusing on different aspects of the scheme). The main additional feature of a level of abstraction is the attempt to provide a consistent layer, comprised of several individual abstractions, that corresponds to a conceptual level of understanding on each side of the interface (with no cross-references between the levels, apart from the lower one implementing the constructs of the higher one). High-level programming languages are a common example of this: programmers write their programs in terms of the language constructs without caring for the implementations of these constructs and language implementors provide such implementations without caring for programs written using these language constructs. It is the task of language designers to define a suitable level of abstraction as an interface between the two separate views. This description supports and refines our earlier claim that modular programming and high-level languages have similar aims – they even use similar means. However, the description in terms of a process of abstraction also highlights several possible problems, concrete examples of which have been documented in the literature.

The problems result from the attempt to treat the two parts of software, high-level programs and language implementations, in isolation. We have already noted that this extreme view would only be possible if the parts were independent,

but even a restricted separation, based on a small interface, will not always work here. The interface between programs and implementations is the language definition: many programs do not depend on the details of the implementation, and general implementations can be provided without taking specific applications (programs) into account. But in general, the suitability of software composed of programs and implementations for a given high-level language may well depend on complex interactions between both components: certain programs depend on time- and space-efficient implementations of language constructs which, on the other hand, can only be provided with certain assumptions about the kind of programs in which these language constructs will be used. The question whether or not programs written in high-level languages can make efficient use of the available resources is probably as old as the languages themselves, but there are also some more specific observations about the interface problem, some of which are addressed below.

One of the earliest observations is that one predefined level of abstraction for a general purpose language may not suit all needs and that a natural way to tackle problems in complex domains would start with the definition of a domain-specific, or even application-specific language. To avoid the duplication of effort involved in the design and implementation of countless individual languages, several proposals have been made, ranging from extensible languages and implementations [Sch71] to the definition of domain-specific languages in general purpose languages, with the simulation classes of Simula 67 [DMN70] being an early example. Our language design aims at the latter approach but takes it further to the level of program components in that neither modules nor objects, nor any other special purpose constructs for modular programming are introduced as language primitives, i.e., the conceptual level of the language constructs is kept rather elementary, but flexible and general enough to enable the problem-free introduction of more specific higher levels.

This aspect of the design is also an attempt to address, at least at the level of modular programming, a problem first noted by the Alphard designers [SW80]. They argued that language designers, in fixing the interface between programs and implementations, tend to pre-empt design decisions that could better be made by programmers: when writing programs in top-down fashion, programmers start with abstract ideas and continue to refine them until every abstract concept has been implemented in terms of the available resources. The implementations are usually chosen so as to take the requirements of the context into account, but this scheme breaks down as soon as the process of refinement reaches the ground level of the programming language: there are still a lot of decisions the programmers would want to make (in contrast to those decisions they do not care about, where the high-level language really frees them from irrelevant details), but everything below this level has been decided by the language implementors. The implementors probably used suitable assumptions for the general case, but they had no chance to take into account the specific needs of our programmers

for the particular problems they are trying to solve (cf. also [Low78]).

The definition of language constructs for modular programming in terms of more elementary constructs allows programmers to use pre-defined abstractions as long as this suits their needs, and it allows them to adapt decisions below this level of abstraction whenever they should need to do so (a similar approach led to the definition of meta-object protocols for CLOS, the Common Lisp Object System [KdRB91, DG87]). Also, new kinds of program components can be defined if necessary, and different kinds of components can be mixed in a single program (e.g., one can collect classes in modules). Translating this approach to the general case of high-level languages requires the provision of elementary and general constructs for all purposes, to be used either directly or indirectly via additional abstractions defined on top of them. It remains to be seen whether all important implementation decisions can be represented in terms of a language level below the real user language without forcing programmers to think about irrelevant details, but if this should work out well, both parts of the abstraction would be available to programmers. This would also solve the problems recently discussed in [Kic92], namely that 'black-box' abstractions (following Parnas' principle of information hiding: the box can be used as it is, but it is not possible to look inside) are not always adequate. There are cases when clients need control over the implementation, i.e., they still want to use the abstraction but also need to make sure that the implementation fits their needs. This has led to a search for similar problems and to the proposal of open implementations (in contrast to the closed black boxes) as a possible solution (cf. [Xer96] and also the work on aspect-oriented programming [KLM+97]).

A classification of abstractions for modular programming

Since we view modules, objects, etc. as instances of a general scheme of abstraction, it suggests itself to search for relations between these instances. Figure 6.19 is a first attempt to classify some abstractions for modular programming from a function-oriented and an object-oriented view. The very first step is to factor programs into functions and parameters (we restrict ourselves to one-parameter functions here). The dichotomy then originates in the abstraction of the algorithm: if it is abstracted with the functions, we get the function-oriented view in the lower left corner of the picture, if it is abstracted with the objects, we get the object-oriented view in the lower right corner. The simplest next step is to bundle several abstractions together, yielding either modules consisting of several function definitions or collections consisting of several object definitions. Objects generally respond to several messages with different methods, implementing several functions in one message-generic object. The counterpart in the function-oriented view are functions applicable to different parameters, called data-generic functions here, which execute one of several different algorithms based on a case analysis of the parameter. In typed function-oriented languages, it is common

practice to classify related objects into types and to speak of functions working on objects of certain types. The object-oriented counterpart to a type would thus be a classification of functions[6] and, indeed, there is such a notion, often called interface.

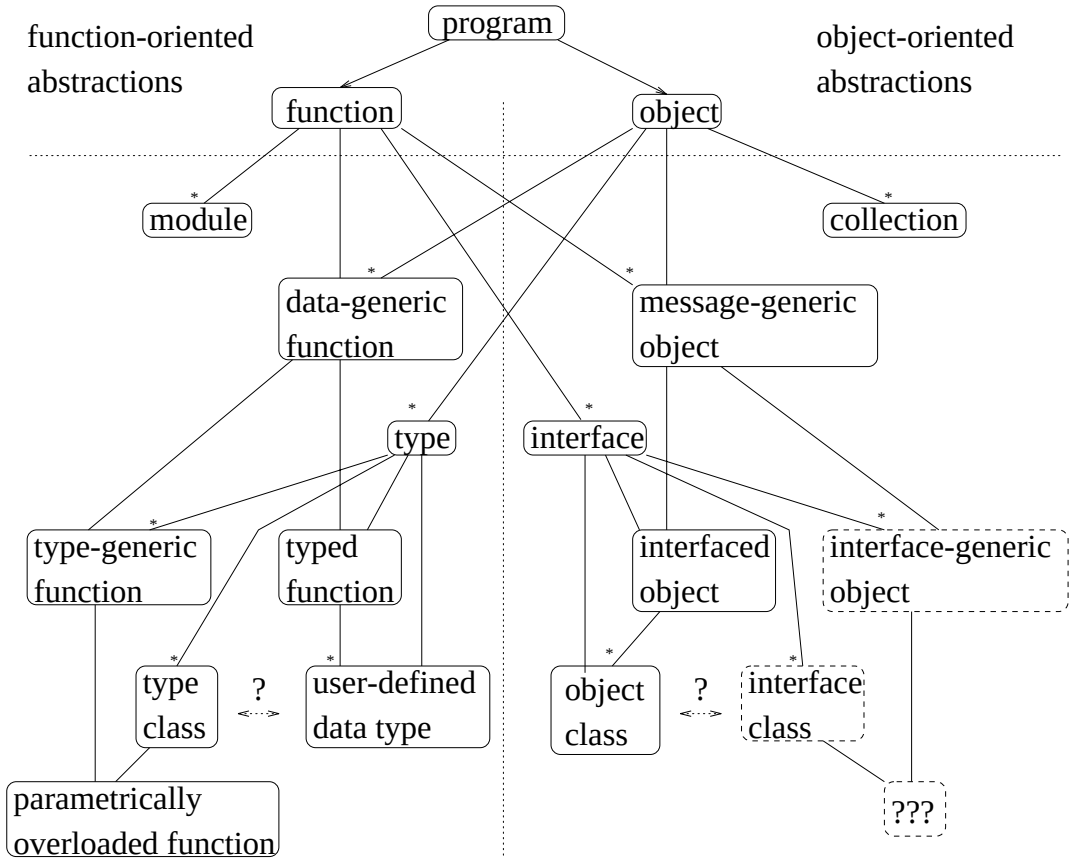

Figure 6.19: Classification of abstractions

Just as a type specifies the acceptable parameters for a function, an interface specifies the acceptable messages for an object, i.e., we can now attribute interfaces to objects. A classification of objects according to their interfaces yields the well-known concept of object classes, where a class consists of several objects sharing one interface - all objects in a class accept the same messages. Again, we can try to find a counterpart for this abstraction on the function-oriented side, a class of several functions accepting the same type of parameter, and this is just a primitive variant of user-defined data types in the sense introduced by [Zil73] and further explored in Russell [DD85]. A new type is defined by providing functions that interpret an existing type. If we would package the latter type with the

[6]Since the algorithm description is kept with the object, functions degenerate to messages in this view.

functions defined on it, we would make the step from user-defined data types to data abstractions as introduced in CLU [LZ74]. The basic distinction between the two concepts is that Russell's user-defined types are interpretations of one universal value spaces whereas CLU's abstract data types assume the set of values used for representations to be typed, too. Hence, data abstractions need to be explicit about the existing type which is interpreted by the functions in the abstraction.

We do not need to stop here: for instance, we could generalize from data-generic to type-generic functions, i.e., functions which accept values of several types and select the actual algorithm to execute according to the type of the parameter. Types can then be classified, leading to Haskell's [PH96] type classes[7], and one generic function will only accept values of types belonging to one type class - Haskell's parametrically overloaded functions. The object-oriented counterpart to type-generic functions seem to be interface-generic objects, objects responding to different interfaces. However, there seems to be no such concept and the same holds for the idea of interface classes and objects responding only to interfaces belonging to one interface class. Since the strategy of searching for counterparts in our picture has otherwise been quite successful, it may be interesting to look for sensible interpretations of these new concepts. If we insist on relating interfaces to object classes, an object responding to several interfaces would necessarily belong to several classes. Note that this is a relation between objects and classes, different from multiple inheritance which is a relation between classes alone. The question marks stand for another, as yet unnamed concept of objects that respond to several interfaces all belonging to the same interface class. We distinguish between object classes and instances of interface classes although both may be just different views of the same concept. A similarly close relationship can be found between instances of type classes and user-defined data types.

This is only a first, rather simplistic attempt of a classification, based solely on the way in which abstractions are constructed, and it certainly needs to be refined and extended (e.g., sub-types and sub-classes are not covered at all). Still, it has some interesting aspects. First of all, it is surprising that it was actually possible to build such a symmetric picture. For each abstraction used in one of the two views, there seems to be a counterpart in the other view, constructed by similar steps from different starting points. Most of these counterparts correspond to known abstractions, though some of them are not usually named, and for those counterparts that are not yet known, reasonable interpretations seem to exist. Another important point is that the abstractions developed for the function-oriented and for the object-oriented view can be divided into two groups. While there is a symmetry between these groups, this symmetry is based on the way abstractions are constructed, not on their interpretations. These in-

[7]Instances of Haskell's type classes can also be seen as user-defined types.

terpretations are different for the two groups and it seems to make no sense to mix abstractions that do not belong to the same group. For instance, the concept of data types belongs to the function-oriented view, but it is rather difficult to ask for the type of an object in the object-oriented view. As the picture suggests, it is more appropriate to ask for the interface accepted by an object. Similarly, object classes classify objects according to their interfaces, whereas data types are classified in type classes. Interestingly, most object-oriented languages depart from a purely object-oriented view to allow for methods with multiple parameters, which leads to a complex mix of object-oriented concepts (classes of objects) and functional concepts (types of methods). The only purely object-oriented language we currently know of is SELF [US87].

Of course, one could introduce a meta-level and view the abstractions in the two lower parts of figure 6.19 as objects of a new programming language. It would then make sense to ask for the meta-type of an interface or for the meta-type of a module. Meta-functions could operate on modules, Classes could be grouped into meta-classes, types into meta-types (usually called kinds). In conventional languages, the step into the meta-level is actually carried out earlier, and types, modules or classes do belong to (different) meta-languages. Since more than two levels of a hierarchy of meta-levels are seldomly used, elements of the three levels are usually named explicitly (e.g., values, types and kinds in the type hierarchy). However, even a single hierarchy of meta-levels gets complex very easily, and the possible combinations between multiple hierarchies tend to aggravate this problem. For instance, consider modular programming in the function-oriented view: should a language first be extended with a type system and then with a module system, or should the module system come before the type system? In the first case, modules would naturally contain types, whereas in the second case modules would naturally have types. The process will usually be repeated to find an additional type system for modules containing types or an additional module system to organize types of modular programs, and the resulting language designs will not only be very complex, they may very well differ for the two different starting points. This is another reason for keeping abstractions first-class language objects: meta-level abstractions can be more expressive than simple abstractions, but as long as this additional expressiveness is not needed, meta-levels should be avoided. To relate this to the given example problem, we have started to add modules to the language, postponing the treatment of types, but since first-class data structures can be used for modular programming, we have not needed a meta-level for modules.

Chapter 7

Implementation

A programming language provides an interface between two conceptual levels of software development. It allows software to be factored into high-level programs using constructs of the programming language and low-level implementations of these constructs. The language specification given in chapter 5 defines the interface between programs and implementations of our language, and some aspects of the high-level view of the language have been developed in chapter 6. The present chapter is dedicated to the low-level view and thereby completes the presentation of our language design. The existing implementation of the core language is not part of the present work, but sets up the framework for the implementation of the new language constructs, so that a summary account of the general implementation is provided here. More detailed accounts of the implementation and of the reduction system itself can be found in [Gär91, Rat97, GK96, Klu94]. For completeness, the path from our description of the language design to the current implementation is outlined in terms of the major implementation decisions involved. The focus is then on the implementation of our language extensions in the existing framework, especially on those aspects that may be relevant to implementations of other functional languages as well.

7.1 Deriving an implementation of the functional core language

In order to build an implementation of the language defined in chapter 5, it is necessary to specify the intended uses of the language, because there is nothing to implement about a definition. In this sense, the only implementation of a programming language worth this general name would be one that provided for all possible uses, which would be quite an ambitious goal. Instead, we restrict our attention to certain uses of the language definition and to the implementation of these. To recall, the language definition itself consists of a description of admissible programs (expression syntax) and a set of (mostly contextfree) transformation

rules which allow both an operational and a declarative interpretation.

First of all, we need to be able to construct and edit programs (expressions) of the language and, further, we want to feed programs into an execution phase and observe the results. Since program execution is defined as program transformation according to the reduction rules of the language, results of program executions are programs, too, and both execution results and sub-expressions may be subjected to (further) reductions. Therefore, a tight integration of the program editing and execution phases is advisable, and the editor should be aware of the language syntax. The editor is used both to enter and to manipulate programs and to display results of program executions and thus becomes the user interface to the implementation. In view of these requirements, the string representation used to display programs to users does hardly suffice – internally, a more structured representation of expressions is needed, together with a representation of the contexts in which expressions and reductions can occur. The representation of expressions follows the tree-shaped abstract syntax used in chapter 5, and a stack is used to keep representations of contexts while traversing sub-expressions. The editor provides for the conversions between the external and the internal representation.

Given representations of expressions and contexts, the next step is to find an implementation of reductions in terms of these representations. The simplest case to consider is the execution of a single reduction step, which means to find an applicable rule and to replace the left-hand side with the right-hand side. There is only one major reduction rule here, namely β, but this rule is context-free and abstracts from the sub-expressions that form the body of the λ-abstraction and the operand. To account for these two generalizations, suitable instantiations for the context and for the two sub-expressions have to be found before the rule can be applied to an expression. In other words, the given expression has to be searched for redices (*redex*: reducible expression). Reducing any of these redices leads to a new expression with a new set of redices. Each expression may thus be seen as the root of a tree, and different paths in the tree correspond to different reduction sequences. A very crude approach to program execution could generate all possible reduction sequences and search the resulting tree for nodes which represent program results (a kind of normal form of the root expression). Provided that a complete search strategy exists, this would be an effective, though not efficient way to implement the language, but fortunately, there is no need to actually generate a representation of the search tree: due to the Church-Rosser property of β, redices may be reduced in any order without affecting the final result. From any node in the tree, there is a path to a result if and only if there is such a path starting with the root (the original program). The partial order of reduction steps (reduction strategy) may, however, have an influence on the use of resources (number of steps/size of expressions), including the extreme case that some strategies may not find existing solutions with finite resources. Beyond these general aspects that can be discussed directly on the basis of the

formal definition of reduction, there are several implementation-specific issues of reduction strategies, but the discussion of these issues has to be postponed until the necessary foundations have been specified. In particular, the implementation of reduction steps and the ensuing refinements of the representation need to be investigated first.

Since redices can occur in operand expressions of other redices, they may be objects of substitutions, and as such they may be duplicated or consumed, depending on the number of occurrences of the variable to be substituted. Duplication of redices may or may not lead to duplication of work (in terms of reduction steps), depending on how many of the duplicates need to be reduced to reach a normal form. On the other hand, reducing redices prior to substitution may turn out to be unnecessary work if the reduced expressions are not used, and it may or may not pay off in terms of space usage (the reduced expressions are not necessarily smaller than the redices). These problems can be avoided if the implementation does not prematurely produce copies of the representations of expressions that are duplicated during substitutions. Instead, copying should be delayed until a modification to only one of the duplicates is necessary or until all reductions that are common to all duplicates have been performed. To achieve these effects of delayed copying, all duplicates of one expressions have to share a single representation, which means that the general representation of expressions has to be adapted: from a tree to a graph. Shared representations do then have a natural implementation as edges to the same sub-graph.

Duplication of operand expressions is not the only feature of β-reduction that deserves special attention in the implementation. Conceptually, each β-reduction step requires a complete traversal of an abstraction body and of an operand to accomplish the substitution and to adapt the number of protection keys of variable occurrences inside the operator and operand expressions. Implemented naively, such a traversal would generate a modified copy of the abstraction body, only to replace the representations of one bound variable with links to a representation of the parameter. If, instead, substitutions are delayed, it is possible to integrate the necessary copying with the traversal that evaluates the instantiated abstraction body and thus to amortize the cost of the copying traversal over several substitutions. The disadvantage is that explicit representations of delayed substitutions have to be constructed in terms of environments binding variables to values, and that the combined traversal has to look up variable bindings in these environments whenever it encounters a variable occurrence. Due to the lexical scoping discipline, positions of variable bindings for dynamic environment lookups can be statically computed, so that each dynamic lookup can be implemented as an indexed access. For instance, expressions with named variables can be converted into expressions in the namefree λ_{NF}-calculus and the deBruijn-indices can be used to access a stack of variable bindings at runtime (the topmost value on the stack always belongs to the innermost variable binding for the current sub-expression). The conversion to the namefree λ_{NF}-calculus takes place in a

preprocessing phase and needs to be reversed in a corresponding postprocessing phase to create the effect of executing β-reduction steps at the user-level. Because the conversions affect neither the structure of expressions nor the transformation behavior, these pre- and postprocessing phases pose no problems, only the names of variables have to be preserved during the processing phase (e.g., as labels of nameless abstractions) to enable a complete reconstruction of (partially) evaluated expressions. In the following, the implementation is therefore described in term of λ_{NF}, with nameless abstractions (Λ) and reductions (β_{NF}).

The graph representation of expressions should allow a simple and inexpensive implementation of β_{NF}-reduction if substitutions are delayed and if only pointers to operand terms need to be copied into and from environments. The major remaining obstacle to such a simple implementation results from the need to adapt protection keys in order to keep the static binding structure intact. This is a fundamental operation that, in general, cannot be avoided without restricting the language, but it is possible to organize reductions in such a way that a full implementation of β_{NF}-reduction is only used if the redex demands it. For a large class of redices, a naive implementation of β_{NF}-reduction (which only substitutes pointers) suffices and is more efficient than the full implementation. To begin with, modifications of protection keys become necessary only if relatively free occurrences of variables are substituted into the scope of other bindings or if β_{NF}-reduction removes a level of variable bindings from an expression containing protected occurrences of variables. Both cases require a search for relatively free or protected occurrences of variables in complete expressions hidden behind pointer abstractions. Since substitutions are delayed until expression evaluation, both problems occur only if bodies of Λ-abstractions are evaluated: either the Λ has just been removed from the abstraction body by a β_{NF}-reduction, unveiling protected variable occurrences inside, or variable occurrences in the values of pending substitutions may be transported into the scope of the Λ. Actually, the first alternative is not a problem with the chosen environment implementation: while the Λ has been removed, the operand expression has been placed on the stack, so that protected variable occurrences in the abstraction body still have to reach beyond the innermost binding level. Environment stacks are nothing but compressed representations of static binding structures, filled with actual variable values, and if the deBruijn-Indices addressed the correct binding in the original expressions, they also address the correct value on the stacks.

As a further step, note that globally free or protected variables, i.e., variable occurrences that are not bound anywhere inside the entire program, do not have to be manipulated during the processing phase: if they occur in the program result, their status with respect to the entire program remains unchanged, so the status can be recorded in the preprocessing phase and reconstructed during the postprocessing phase, whereas the processing phase can treat them as constant objects. The problem of relatively free variable occurrences can thus be avoided by simply not evaluating the bodies of Λ-abstractions until the abstractions are

paired with operands to form β_{NF}-redices. This restricts the possible reduction strategies to a top-down approach, and since globally free variable occurrences are not an issue anymore, and since no reductions are done inside of abstractions, it seems as if all relatively free variable occurrences would be substituted by values before they could participate in further reductions. However, this scheme can only implement a reduction to weak head normal forms, and abstractions which are not in operator position of redices are neither evaluated nor are any pending substitutions performed on them. As a consequence, such abstractions need to be paired with copies of their environments for later access, forming so called *closures*.

The environment stack can become quite large during evaluation, even though each individual sub-expression usually need access to a few entries only, and it would therefore be expensive to store a copy of the complete current environment into each closure. Since the indices by which the environment will be accessed are known statically, it is possible to pre-compute, for each abstraction, the sequence of relevant entries from the current environment and to copy only this condensed environment into the closure. Of course, variable occurrences in the each abstraction body need to be adapted to refer to the condensed environment (the variable bindings in the closure), which provides bindings for all but the globally free variable occurrences in the abstraction body. Such an abstraction with a closed local environment can be represented as a Λ-expression (the relatively free variables are passed as additional parameters to the abstraction), so that no special runtime mechanisms are needed to express the environment manipulations. The additional abstractions and applications are introduced in the preprocessing phase and need to be tagged to distinguish them from the original program constructs. Tagged constructs that are not evaluated in the processing phase are eliminated by the postprocessor. If all abstractions in an expression are closed, the abstractions are also called *supercombinators*.

Supercombinators are known to have advantageous properties with respect to an efficient implementation of reduction. First and foremost, substitutions never need to cross abstractions because there are no relatively free variables in them. Operationally speaking: substitutions result from reducing β_{NF}-redices, and whenever a substitution encounters an abstraction, it is converted back into a new β_{NF}-redex. The redex itself is the result of statically closing the original abstraction, and the substitution fills in the actual values for the parameters at runtime. However, to take advantage of this property of supercombinators, it is also necessary to restrict the reduction strategy, as simply reducing the additional redices in isolation would instantiate new substitutions and cause them to cross the original abstractions. Therefore, redices with supercombinators in operator position can only be reduced if all parameters are available, and all nested redices of such a fully applied supercombinator redex have to be reduced in one step (*supercombinator reduction*). Due to the statically pre-computed closures, supercombinator reduction is an efficient reduction scheme, but it implements

reduction to weak head normal form only – bodies of (partially applied) abstractions are still not evaluated.

Fortunately, a simple trick can be employed to combine supercombinator reduction with reduction to normal form. To understand this, note that bodies of (partially applied) abstractions need not be evaluated in the processing phase. While it may occasionally be useful to partially evaluate abstractions (using full β_{NF}-reduction) before they are used as operators, the decision to postpone the evaluation until all parameters are available can usually be justified by efficiency considerations. So, supercombinator reduction fails only if (partially applied) abstractions are part of the reduction result. This happens if these abstractions are never applied to further parameters, i.e., the context in which the abstraction is embedded is constant – it contains no further redices. Now, the trick is to identify such abstractions in the postprocessing phase, and to call the processing phase again with the bodies of these abstractions. To do this, the partial application is η-expanded to construct a full application inside a new abstraction.

$$\begin{aligned}&\forall v_i \in var,\ r, n \in Nat,\ r < n :\\ &\qquad (\lambda(v_i)_{1\le i\le n}.expr\ (expr_i)_{1\le i\le r})\\ &\qquad =_\eta\\ &\qquad \lambda(v_i)_{(r+1)\le i\le n}.(\lambda(v_i)_{1\le i\le n}.expr\ (expr_i)_{1\le i\le r}\ (v_i)_{(r+1)\le i\le n})\end{aligned}$$

The new abstraction has only the missing parameters as formal parameters, and the original abstraction is applied to these to form a full supercombinator redex. Since only the body of the new abstraction is subjected to further reductions, the additional parameters are treated as globally free variables (see above).

With these details in place we can now return to the issue of reduction strategies. Normal order reduction, even if combined with sharing to yield lazy evaluation, incurs an overhead with respect to delayed substitutions. Instead of actually evaluating each expression in the current environment, evaluation is delayed until the latest possible point in the reduction sequence. This means that representations of yet-to-be-evaluated expressions have to be created, and since pointers to these representations will be substituted into contexts where the original variable environment is no longer directly accessible, each of them has to be paired with a representation of its environment, forming lots of unevaluated *closures*. The costs of building and managing these closures instead of just evaluating the expressions are one of the main reasons why applicative order reduction is the standard strategy in our reduction systems (the other reason is its suitability for non-sequential evaluation). This decision is obviously based on implementation (efficiency) considerations and emphasizes that the prevalent focus of research in our group has not been on language design issues so far, but rather on the design of hard- and software architectures to support a high-level functional programming paradigm. The decision can be justified at the user level, too, because (a) the availability of efficient implementations is an important issue for the acceptance of a paradigm and the usability of its languages, and because (b) even

an applicative order λ-calculus is still computationally complete. In an overall view of software systems (programs in high-level language plus implementation thereof), an applicative order reduction strategy simply shifts some responsibility from the implementation, where it cannot currently be handled completely satisfactory, to the high-level programs, which sometimes have to be reformulated to take the problems of applicative order reduction into account. From a purely high-level perspective, however, it would be preferable if user programs would not be burdened with these efficiency considerations, especially since applicative order reduction restricts the validity of β-equivalence (and thus the means for abstraction in our language design!) to expressions that have a value under this strategy. Consequently, some of our reduction systems optionally support normal order reduction or lazy evaluation, but for simplicity we stick with applicative order reduction here.

$$
\begin{array}{lcl}
caf_{whnf} & = & i \mid (caf_{whnf}\ whnf) \\
whnf & = & \Lambda.\ expr \mid caf_{whnf}
\end{array}
$$

$$
\begin{array}{lcl}
aoc & = & [\,] \mid (expr\ aoc) \mid (aoc\ expr) \\
rc & = & [\,] \mid (expr\ rc) \mid (rc\ expr) \mid \Lambda.\ rc
\end{array}
$$

$$
\begin{array}{lcl}
aoc[(\Lambda.expr\ whnf)] & \mapsto_{\beta_{NF},aor} & aoc[\Pi_1^{-1}\ expr[0 \leftarrow \Pi_0^{+1}\ whnf]] \\
rc[(\Lambda.expr\ whnf)] & \mapsto_{\beta_{NF},aor} & rc[\Pi_1^{-1}\ expr[0 \leftarrow \Pi_0^{+1}\ whnf]]
\end{array}
$$

Figure 7.1: Weak head normal forms and applicative order reduction contexts

Basically, applicative order reduction requires inner redices to be reduced before outer ones. Figure 7.1 defines weak head normal form ($whnf$), applicative order reduction contexts (aoc), and applicative order reduction ($\mapsto_{\beta_{NF},aor}$) for the λ-calculus fragment of our language. Note again that reduction in applicative order contexts aoc (to weak head normal form) realizes only a subset of reduction in all contexts rc (to normal form). These definitions can directly be translated into a recursive algorithm (figure 7.2), searching an expression for redices and reducing them. The two-phase scheme accounts for the two kinds of reduction contexts, rc and aoc (red_{nf} treats redices left over by aor). For the main part of the algorithm (aor), each expression is either in weak head normal form, or it has an applicative order reduction context, which can be the empty context or an application, leading to three rules [1]. Essentially, reduction contexts have been refined to a recursive search, and only sub-expressions that might become new redices are searched anew after reductions. The explicit use of the syntactic category $whnf$ in some of the rules would require complex recursive tests in practice but can be avoided easily: the revised main part of the algorithm in figure 7.3 takes into account that its own result is always in $whnf$. Two rule sets are used to distinguish between those applications for which it is known that operand

[1] The rules have to be applied in sequence because some of the left-hand sides overlap.

$$
\begin{array}{lcl}
red\ M & = & red_{nf}\ (aor\ M) \\
red_{nf}\ (M\ N) & = & (red_{nf}\ M\ red_{nf}\ N) \\
red_{nf}\ \Lambda.M & = & \Lambda.\ red\ M \\
red_{nf}\ i & = & i \\
aor\ whnf & = & whnf \\
aor\ (\Lambda.M\ whnf) & = & aor\ \Pi_1^{-1}\ [0 \leftarrow \Pi_0^{+1}\ whnf]M \\
aor\ (M\ N) & = & aor\ (aor\ M\ \ aor\ N)
\end{array}
$$

Figure 7.2: Direct recursive algorithm

and operator expressions are in $whnf$ (rule set ap) and those for which this may not be true (rule set aor). As the names indicate, aor searches an expression for potential redices and ap examines applications after their sub-expressions have recursively been reduced to weak head normal form. If an application is a redex, it is reduced and the result is passed to aor again.

Current stock hardware (or the abstract view of it as presented, e.g., by C if it is used as a portable assembler) does not support the general form of recursion used in figures 7.2 and 7.3 very well, necessitating further transformations of the algorithm to bring it into an iterative (or *tail-recursive*) form. Such a form lends itself to an implementation in terms of a control loop manipulating the state of a store. There are two kinds of recursion involved: the iteration of reduction over reduced expressions and the recursive descent into sub-expressions that corresponds to the recursively defined reduction contexts. These recursions can be merged into a single iteration if sub-expressions are treated in sequence, and if an explicit representation of contexts is introduced. There are two standard techniques for the latter, either using a *control stack* to represent context information or transforming the whole expression into *continuation-passing style (CPS)*. CPS-transformations explicitly encode an evaluation order and the original tree-shaped expressions into expressions of a linearly recursive structure that can be evaluated iteratively. To this end, the residual of evaluation after each reduction step is encoded as an explicit continuation function which gets the result of the step as its parameter. The stack-based approach, on the other hand, encodes part of the recursive control structure of an algorithm in form of a recursive data structure, the control stack. The iterative algorithm in figure 7.4 shows the result of using this approach on the recursive reduction algorithm. Again, there are two sets of rules: $aor \downarrow$ describes the search for redices in sub-expressions, building a stack S of contexts, and $aor \uparrow$ reduces redices and reconstructs expressions from reduced sub-expressions and contexts on the stack. The order of reduction has been sequentialized to reduce operands before operators, leading to an additional rule in $aor \uparrow$ (the first one) to switch from the evaluation of an operand expression to an expression in operator position. This abstract description of the algorithm makes it obvious that the use of a control stack corresponds closely to construct-

$$
\begin{array}{lcl}
aor\ (M\ N) & = & ap\ (aor\ M\ \ aor\ N) \\
aor\ M & = & M \\
\\
ap\ (\Lambda.M\ N) & = & aor\ \Pi_1^{-1}\ [0 \leftarrow \Pi_0^{+1}\ N]M \\
ap\ M & = & M
\end{array}
$$

Figure 7.3: Recursive algorithm without complex tests for whnf

$$
\begin{array}{lcl}
aor \downarrow\ (M\ N)\ \ S & = & aor \downarrow\ N\ (M\ []) : S \\
aor \downarrow\ M\ \ S & = & aor \uparrow\ M\ S \\
\\
aor \uparrow\ N\ (M\ []) : S & = & aor \downarrow\ M\ ([]\ N) : S \\
aor \uparrow\ \Lambda.M\ ([]\ N) : S & = & aor \downarrow\ \Pi_1^{-1}\ [0 \leftarrow \Pi_0^{+1}\ N]M\ S \\
aor \uparrow\ M\ ([]\ N) : S & = & aor \uparrow\ (M\ N)\ S \\
aor \uparrow\ M\ nil & = & M
\end{array}
$$

Figure 7.4: Iterative algorithm with explicit context stack

$$
\begin{array}{lcl}
aor_p\ (M\ N)\ \ S & = & ap\ (aor_p\ M\ (aor_p\ N\ S)) \\
aor_p\ M\ \ S & = & M : S \\
\\
ap\ \Lambda.M\ :\ N\ :\ S & = & aor_p\ \Pi_1^{-1}\ [0 \leftarrow \Pi_0^{+1}\ N]M\ S \\
ap\ M\ :\ N\ :\ S & = & (M\ N)\ :\ S
\end{array}
$$

Figure 7.5: Partially evaluated algorithm

$$
\begin{array}{lcl}
aor_p\ i\ \ E\ S & = & (lookup\ i\ E) : S \\
aor_p\ (M\ N)\ \ E\ S & = & ap\ (aor_p\ M\ E\ (aor_p\ N\ E\ S)) \\
aor_p\ M\ \ E\ S & = & <M,\ E> : S \\
\\
ap\ <\Lambda.M,\ E_M>:<N,\ E_N> : S & = & aor_p\ M\ <N,\ E_N> : E_M\ S \\
ap\ <M,\ E_M>:<N,\ E_N> : S & = & <([E_M]M\ [E_N]N),\ nil> : S
\end{array}
$$

Figure 7.6: Partially evaluated algorithm – enhanced

$$
\begin{array}{lcl}
aor_p\ i\ \ E\ S & = & (lookup\ i\ E) : S \\
aor_p\ (M\ N)\ E\ S & = & ap\ (aor_p\ M\ E\ (aor_p\ N\ E\ S)) \\
aor_p\ \Lambda.M\ \ E\ S & = & <\Lambda_{aor_p\ M}.M,\ E> : S \\
\\
ap\ <\Lambda_{C_M}.M,\ E_M> : Cl_N : S & = & C_M\ Cl_N : E_M\ S \\
ap\ <M,\ E_M>:<N,\ E_N> : S & = & <([E_M]M\ [E_N]N),\ nil> : S
\end{array}
$$

Figure 7.7: Separated compiler and runtime system

ing continuations on the fly: the context stack holds (most of) the continuation, and the hole in the top-most context on the stack marks the position where the result of evaluating the current sub-expression will be placed.

The first iterative version of the algorithm corresponds to a universal interpreter for reductions: a fixed general control structure interprets representations of expressions and contexts to determine the specific control structure for the actual program expression just before evaluation. There are several opportunities to improve on this behavior by pre-computing the program-specific control structure, e.g., by *partial evaluation*. First of all, the algorithm calls itself with modified parameters, which are then used to direct the subsequent evaluation. While the exact parameters are not known statically, some of the modifications are known (they are specified in the right-hand sides of the rules). Specializing the algorithm with respect to this partial information yields the new variant defined in figure 7.5. This algorithm has two phases, the first of which translates the static structure of the argument expression into a sequence of stack manipulations and applications of the old iterative algorithm to the statically known redices. Since the explicit context information was only used to direct the subsequent evaluation, this information has been eliminated from this version completely. The first phase (aor_p) does not even inspect the stack but uses the statically known structure of the original expression to pre-compute the evaluation order of the algorithm. The second phase (ap), however, still has to inspect the stack contents to decide whether the two topmost entries form a redex or not. If they do, the reductum will be a new expression that does not exist prior to reduction. Therefore, either the interpreter has to be called to further evaluate it, or the second phase has to call the first phase at runtime to pre-compute the next sequence of operations (the latter variant is shown in figure 7.5: apart from the stack, it closely resembles the variant in figure 7.3).

This is still not the end of optimization as the structure of expressions that result from reductions is not taken into account before reduction. While it may seem as if information about these expressions would not be available statically, a closer look at the β_{NF}-reduction rule reveals that the topmost part of the structure of the right-hand side is just taken from the body of the Λ-abstraction on the left-hand side. Since variables are just placeholders for unknown expressions, the body of an abstraction contains partial information about the result of applying the abstraction to an argument. Substitution does not invalidate this information in any way, but refines it by replacing placeholders with known expressions. The enhanced algorithm in figure 7.7 employs this property of β_{NF}-reduction to pre-compute even larger fragments of its own evaluation order for a given program expression statically. Figure 7.6 shows an intermediate version that incorporates the idea of delayed substitution: it uses an additional stack E to represent the current variable environment and looks up bindings of variables when it encounters them during evaluation. Since bodies of abstractions are not evaluated, they need to be paired with duplicates of their lexical environ-

ments, forming closures (for uniform access, every object on a stack is a closure, written as $< expr,\ env >$). Globally free variables are treated as constants to be reconverted to variables later (with the correct number of protection keys). This intermediate variant of the algorithm still alternates between two evaluation phases which resemble a compiler (aor_p, pre-computing evaluation order) and an interpreter (ap, performing reductions). The interpreter seems necessary to handle dynamically constructed applications, to reduce redices and to instantiate constant applicative forms (notation: $[env]caf$), but at least it can call the compiler dynamically to avoid some interpretative overhead.

The important observation to make about figure 7.6 is that the results of the calls to the compiler in the interpreter do only partially depend on the dynamic variable environment. Most of the evaluation order can be pre-computed using only statically known information – lexical scoping determines the presence or absence and even the position of variables in the environment, though not their value bindings, and the decisions to be made in aor_p depend only on the static program structure. If the value bindings of variables and the results of β_{NF}-reductions are left as dynamic inputs to the reduction algorithm, it is still possible to completely separate the evaluation into two phases. The first phase (aor_p, compilation) pre-computes the evaluation order in several segments, each corresponding to the body of one Λ-abstraction, and annotates these Λ-abstractions with partially evaluated functions. The second phase (ap, runtime system) evaluates the argument expression according to the pre-computed evaluation order and the dynamic parts of the algorithm input. Instead of calling an interpreter or compiler at runtime, it uses the code with which Λ-abstractions have been annotated (cf. figure 7.7). The fundamental idea that led to this two-phase algorithm was to identify statically available information in the arguments and the definition of the first iterative reduction algorithm. This first version of the algorithm was then partially evaluated with respect to the static information, moving all aspects of the algorithm that do not depend on information declared as dynamic to a first phase of the transformed algorithm and leaving all other aspects in the residual of the partial evaluation[2].

All that remains to be done is to actually separate the two phases, i.e., to define the capabilities of the runtime system and to remove the dynamic parts of the input (workspace stack S and environment contents E) from the compiler definition (the current size d of the environment is still needed to identify globally free variables). Figure 7.8 outlines the final version of compiler and runtime system (still simplified, but complete with reduction to normal form and treatment of globally free variables) and completes the description of the implementation at this level of abstraction. A more conventional representation of compiled code in terms of sequences of instructions is used, and compiled code is attached to Λ-abstractions. Variable environment E and workspace stack S are combined

[2]Partial evaluation is a standard program transformation technique [CD93].

$$
\begin{array}{lcl}
red\ d\ E\ M & = & red_{nf}\ d\ E\ (exec\ (aor_p\ d\ M)\ E) \\
red_{nf}\ d\ E\ (M\ N) & = & ((red_{nf}\ d\ E\ M)\ (red_{nf}\ d\ E\ N)) \\
red_{nf}\ d\ E\ (\Lambda_C^{n+1}.M\ (E_i)_{1\le i\le n}) & = & \Lambda.(restart\ (d+1)((E_i :)_{n\ge i\ge 1}\ E)\ C) \\
red_{nf}\ d\ E\ \$i & = & \#(i+d) \\
red_{nf}\ d\ E\ \#i & = & \#i
\end{array}
$$

$$
\begin{array}{lcl}
exec\ C\ E & = & strip\ (C\ <E, nil>) \\
strip\ <E, M : S> & = & M \\
restart\ d\ E\ C & = & (red_{nf}\ d\ (\$(\Leftrightarrow d) : E)\ (exec\ C\ (\$(\Leftrightarrow d) : E)))
\end{array}
$$

$$
\begin{array}{rcl}
f\ ;\ g\ <E,\ S> & = & g\ (f\ <E, S>) \\
pushArg\ i\ <(E_i)_{i\in I},\ S> & = & <(E_i)_{i\in I},\ E_i : S> \\
pushClos\ \Lambda_C^{n+1}.M\ <E,\ (E_i :)_{n\ge i\ge 1}\ S> & = & <E,\ (\Lambda_C^{n+1}.M\ (E_i)_{1\le i\le n}) : S> \\
pushConst\ c\ <E,\ S> & = & <E,\ c : S> \\
popE\ n\ <(E_i :)_{n\ge i\ge 1}\ E,\ S> & = & <E,\ S> \\
beta\ \Lambda_C^{n+1}.M\ \ <E,\ N : (E_i :)_{n\ge i\ge 1}\ S> & = & C;\ popE\ (n+1) \\
 & & <N : (E_i :)_{n\ge i\ge 1}\ E,\ S>
\end{array}
$$

$$
\begin{array}{lcl}
aor_p\ d\ \#i & = & pushArg\ i \qquad ,\ if\ i < d \\
aor_p\ d\ \#i & = & pushConst\ \$(i \Leftrightarrow d)\ \ ,\ if\ i \ge d \\
aor_p\ d\ ((\Lambda^{n+1}.M\ (\#i_j)_{1\le j\le n})\ N) & = & (pushArg\ i_j :)_{1\le j\le n}\ aor_p\ d\ N; \\
 & & beta\ \Lambda^{n+1}_{aor_p\ (d+n+1)\ M}.M \\
aor_p\ d\ (\Lambda^{n+1}.M\ (\#i_j)_{1\le j\le n}) & = & (pushArg\ i_j :)_{1\le j\le n} \\
 & & pushClos\ \Lambda^{n+1}_{aor_p\ (d+n+1)\ M}.M \\
aor_p\ d\ (M\ N) & = & aor_p\ d\ N;\ aor_p\ d\ M;\ ap
\end{array}
$$

$$
\begin{array}{lcl}
ap < E, (\Lambda_C^{n+1}.M\ (E_i)_{1\le i\le n}) : N : S > & = & C;\ popE\ (n+1) \\
 & & <N : (E_i :)_{n\ge i\ge 1}\ E, S> \\
ap < E, M : N : S > & = & < E, (M\ N) : S >
\end{array}
$$

Figure 7.8: Final version of simplified compiler and runtime system

to form an abstract machine state for the runtime system, which is installed and destroyed in *exec*. Note that the explicit closures of the intermediate variants (cf. figures 7.6 and 7.7) are replaced by partially applied supercombinators (*pushClos*), and that existing code can be used when the reduction of partially applied supercombinators is restarted after η-expansion in red_{nf} (calling *restart* instead of *red*). It is assumed here that each Λ-abstraction is closed individually during preprocessing, hence each former one-parameter abstraction is modified to take n additional parameters (which supply values for the n relatively free variables of the original abstraction). Of course, multi-parameter abstractions are closed and reduced as a whole in the full system.

The implementation of reduction has so far been described in terms of a graph representation but, on conventional hardware, the graph has to be mapped to random access memory. Essentially, each node in the graph represents a constructor of the language (cf. the abstract syntax defined in chapter 5). It carries information about the language construct (abstraction, application, ...) and links to the representations of the sub-expressions. A natural representation of a node uses a node-dependent number of consecutive memory cells, each containing a part of the node information or a pointer to the representation of another node (we call such sequences of cells *heap objects* from now on). Initially, heap objects can simply allocate cells from the beginning of the free memory area, but if memory requirements go beyond the finite amount of available memory, representations of nodes that are no longer used need to be deallocated. To this end, the available memory fragments are registered in a *free (memory) list*, and new objects allocate memory from the beginning of the first memory fragment in the free list that is large enough to hold the new object. If there is no such fragment, the memory needs to be reorganized by shifting all live objects to the beginning of memory and by merging all free fragments at the end (*garbage collection*). Such a garbage collection changes the location of objects in memory and thus invalidates any pointer scheme based on these locations. All pointers have to be adapted to point to the new object locations which means that memory has to be reserved for a table associating old locations with new ones. This table could be generated anew on every garbage collection, but can also be used continuously as an indirect pointer implementation. In the case of a permanent indirection table, every pointer to a heap object goes via this table. This suggests a slightly different approach in which every node is represented by a fixed-size entry in a table of *descriptors*. The size of descriptors is chosen so as to accommodate the most important bits of information for each kind of node, and if the amount of information exceeds the descriptor size, the descriptor is supplemented by a variable-sized heap object. Due to the fixed descriptor size, no fragmentation can occur in the descriptor table, and while the heap is still subject to garbage collections, only the corresponding entry in exactly one descriptor has to be updated if the location of a heap object changes.

To keep the overall memory usage small without incurring frequent garbage

collections, memory for unused descriptors and objects has to be deallocated as early as possible. A large step in this direction can be made if each descriptor 'knows' whether it represents a live object or not. To this effect, each descriptor has a field in which the number of references to this descriptor is kept. Whenever a pointer to a descriptor is duplicated or consumed, the corresponding *reference count* is updated, and if it drops to zero, the memory used by the descriptor and its heap object can be deallocated (added to the free list). The need for some kind of reference counting is a consequence of the graph sharing that may result from delayed copying: as long as the reference count of a descriptor is greater than one, a copy operation of the sub-graph is still pending and the shared representation of the original sub-graph must neither be deleted nor modified (unless the modification is valid in all copies). However, reference counting could also be delayed until a garbage collection actually occurs, leaving parts of the memory unused in between, while avoiding some of the time and space costs of continuous reference counting.

This completes the informal derivation of an implementation from the language definition given in chapter 5, presented at a level of detail that suffices for the following discussion of our extensions. Further details of the compilation scheme and runtime system, including optimizations and the treatment of other language elements, can be found in [GK96]. The figures 7.9 and 7.10 outline the general structure of a reduction system, as it results from the derivation given above. Figure 7.9 depicts the major phases of different usage cycles supported by a reduction system, annotated with the abstract representations used for expressions at different levels. Expressions are presented as strings and stored internally as abstract syntax trees which may be subjected to reduction, edited, or stored in the file system. A preprocessor prepares the tree representation for reduction and a postprocessor undoes the representation changes as far as necessary to present reduction results at the user level. The abstract syntax graph is based on name-free expressions, but abstractions are labeled with the original variable names for use in the postprocessor. Compilation converts graphs to code, and decompilation reverses this process to uncover parts of the graph which are hidden in not yet executed code fragments. Basically, the remaining code is executed without performing any further reductions, so that the runtime system only constructs applications. Reduction to normal form inspects the result of execution and may start further code execution phases to evaluate bodies of (partially applied) abstractions. It is thus directly connected to the runtime system.

Figure 7.10 summarizes the memory structures used for the low-level representations of graphs and contexts. The runtime system is controlled by the code and accesses the graph representations via a system of stacks. The argument stack A holds the variable environment, the workspace stack W holds temporaries, and stack R holds return addresses during function calls (R was not introduced in the derivation given above but is simply used to make the recursion in the compiled code explicit). The central component of the memory system is the descriptor

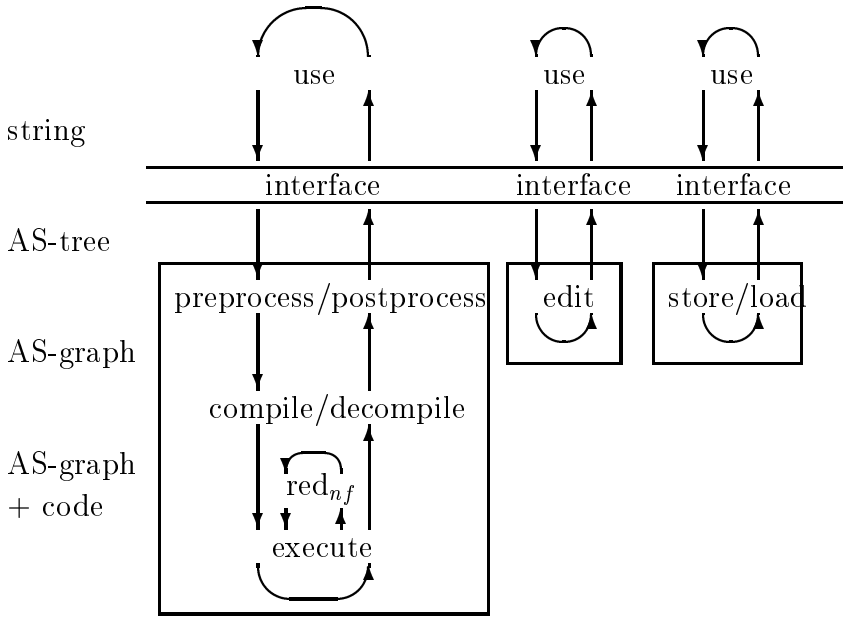

Figure 7.9: Components, representations, and phases of a reduction system

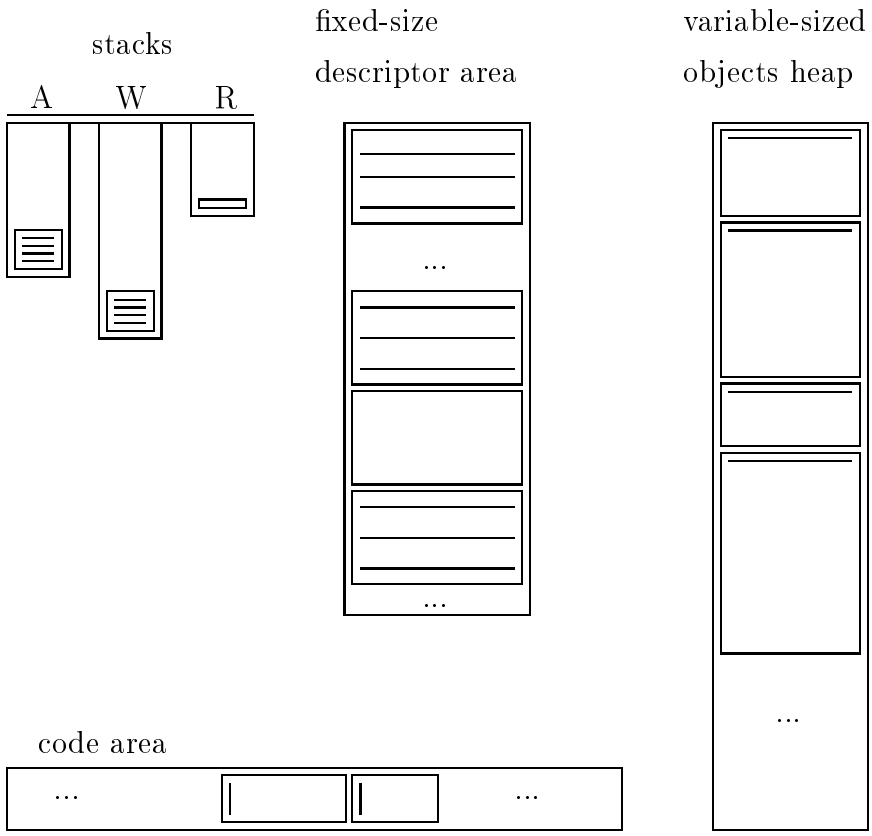

Figure 7.10: Memory structures of a reduction system

area because, in principle, every pointer between memory structures goes via a descriptor. There are about 60 types of descriptors representing different syntactic categories and even more different stack elements, which are both used to keep small data objects directly in the runtime stack system and for the abstract syntax tree representation of expressions (unfortunately, this representation is also used for some kinds of irreducible expressions in the runtime heap). Furthermore, about 200 different instructions may occur in the code region, which is also special in that it contains direct pointers to code for jumps and function calls (the code area is assumed to be static, i.e., it is not subject to garbage collections and relocations, so that there is no need to adapt pointers).

7.2 Frames

The implementation extensions to support frames are straightforward. Each frame is represented by a descriptor that points to a heap object in which (pointers to) the representations of the slots are stored. The names of the primitive operations (`select`, `delete`, `update`, `test`, `slots`) are introduced as new predefined constants in the compiler, and implementations of the corresponding δ-rules in terms of the graph representation are added to the runtime library (not shown here). The common core functionality of finding a slot with a given name (slot lookup) in a frame is implemented as a linear search over the list of slots. Finally, new instructions are added to construct frames and slots. Figure 7.11 shows the extensions to the abstract specification of the implementation derived in the last section.

$$
\begin{array}{l}
red_{nf}\ d\ E\ \{\ (name_i :: value_i)_{1\le i\le n}\ \} \\
\quad =\ \{\ (name_i :: red_{nf}\ d\ E\ value_i)_{1\le i\le n}\ \} \\
\\
mkSlot\ <E,\ name : value : S> \quad = \quad <E,\ (name :: value) : S> \\
mkFrame\ n\ <E,\ ((slot_i)_{1\le i\le n}\ S)> \quad = \quad <E,\ \{\ (slot_i)_{1\le i\le n}\ \} : S> \\
\\
aor_p\ d\ \{\ (name_i :: value_i)_{1\le i\le n}\ \} \\
\quad =\ (aor_p\ d\ value_i;\ pushConst\ name_i;\ mkSlot;)_{1\le i\le n}\ mkFrame\ n
\end{array}
$$

Figure 7.11: Implementation extensions for frames

Obviously, there are many options for optimizations of this simple implementation: if frames can be expected to have a large number of slots and if structure-preserving operations (select or update a slot) are more common than structure modifications (delete or add a slot), hash tables could be used to speed up slot lookup. If, furthermore, the slot names used in slot lookups are known in advance, the positions of slots could be pre-computed, sharing the costs of slot lookup between several operations on frames of known structure (ideally,

positions for slot lookup could be statically pre-computed similar to positions for environment lookup). Beyond these optimizations of the internal slot lookup, the δ-rules could be used to 'fold away' combinations of operations that cancel each other, e.g., an update followed by a select of the same slot is equivalent to the slotvalue, and a frame which is used only for a known set of selection operations need not even be constructed. On the other hand, frames were introduced to overcome the limitations of static scoping rules, so that slot selection and frame modifications can be expected to be used in not statically predictable ways frequently.

It might seem as if the main intended use of frames as collections of program building blocks would be a serious obstacle to optimization, but this need not be true. If frames are stored and compiled separately from the programs that use them, it is usually not feasible to produce specialized variants of code for all possible kinds of use in advance. The major advantages of optimizations, however, are not bound to the exploration of static knowledge for program modifications before runtime but only to the sharing of costs between several reduction steps (or function calls, ...). So it should still be possible to optimize frame operations when a frame first comes into scope, e.g., through an input/output-operation (this may involve runtime calls to the compiler). But even if such sophisticated optimizations are not implemented, the available language constructs can be used to distinguish clearly between bindings to statically scoped names and selections from dynamically modifiable frames. For instance, imported items have been bound to local names prior to using them in the examples in chapter 6, thus replacing expensive selections with efficient environment access as soon as possible and sharing the costs of slot selection between all uses of the imported items without needing any special optimizations.

7.3 Interactions

Primitive interactions have lots in common with primitive functions, and so it makes sense to implement the former as a special kind of the latter. The major differences between the interaction rules given in the formal language definition (cf. figure 5.8 in chapter 5 or the abstract summary in figure 7.12) and typical δ-rules are the restricted evaluation contexts (either empty or the leftmost path in a tree of `>>=`-applications) and the reference to an entity external to the language (the environment). As far as the implementation of primitive interactions is concerned, the environment is an additional parameter and result value, supplied via additional internal mechanisms. In other words, the runtime library of primitive interactions provides implementations which have to be called using a scheme that is slightly different from that for implementations of primitive functions. For the existing implementation, an interaction is similar to a primitive function that can be applied to parameters but is never evaluated. This

behavior of the existing implementation closely resembles the idea of interactions outlined in chapter 3: a description of the desired interactions is constructed and returned as the result of the functional part of the computation. The addition of input/output-facilities does not affect the functional part of the language, and the implementation of the functional part is not even changed.

$$
\begin{array}{lcl}
ic & = & [\,] \mid (ic \;\; >\!\!>\!\!= \;\; expr) \\
ic[\,((return\ res) \;\; >\!\!>\!\!= \;\; cont)\,] \mid\mid env & =_I & ic[\,(cont\ res)\,] \mid\mid env \\
ic[\,(primInter\ parms)\,] \mid\mid env & =_I & ic[\,(return\ res)\,] \mid\mid env`
\end{array}
$$

Figure 7.12: Interaction contexts and rules

For the actual execution of interaction descriptions, an external interpreter was assumed in chapter 3, and the interaction rules of chapter 5 are defined in contexts where (a representation) of the external environment is available (cf. figure 7.12). The natural first approach to the implementation of interactions is thus to define an interpreter that inspects the results of program reductions (cf. figure 7.13). If this interpreter finds an executable primitive interaction description in a valid interaction context, it provides the implementation of this interaction with access to the program's runtime environment S, replaces the interaction description with the interaction result and calls itself with the modified interaction script. The implementation of the transformation rule for `>>=` and `return` is integrated into the interaction interpreter. The only complication arises from the fact that the result of evaluating a composition of interactions (combined with `>>=`) may be a dynamically constructed reducible application. To evaluate such an application, the interaction interpreter calls the reducer, which may return a new interaction description. In effect, expression reducer and interaction interpreter cooperate as coroutines.

$$
\begin{array}{lcl}
inter\ (primInter\ parms)\ S & = & inter\ (return\ R)\ S'\ , \\
 & & if\ primInter\ parms\ S = (R, S') \\
inter\ ((return\ R) \;\; >\!\!>\!\!= \;\; M)\ S & = & inter\ (aor\ (M\ R))\ S \\
inter\ (N \;\; >\!\!>\!\!= \;\; M)\ S & = & bind\ (inter\ N\ S)\ M \\
inter\ whnf\ S & = & (whnf,\ S) \\
bind\ (R_N,\ S_N)\ M & = & inter\ (R_N \;\; >\!\!>\!\!= \;\; M)\ S_N
\end{array}
$$

Figure 7.13: Interaction interpreter

Similar to the core reduction sytem, the interpreter can be transformed into an iterative form using a stack, and it can be optimized by moving decisions from the runtime system to a preprocessing phase and by avoiding the creation of intermediate structures but, in general, the ability to direct the further evaluation

by an inspection of intermediate graph structures will be a necessary component of the runtime system as long as the language allows to compute new interaction scripts at runtime. Figure 7.14 shows an iterative interpreter variant using a stack of contexts C (note the embedded call to *aor*). This variant corresponds closely to the current implementation (cf. [Tim96]) and keeps the interaction interpreter and the reduction system separated. The interaction interpreter becomes the main entry point for the combined system and calls the reduction system if necessary (fourth rule in figure 7.14). This separation is achieved at the cost of intermediate graph structures for interaction descriptions which are constructed as irreducible applications in the reduction system and analyzed in the interaction interpreter. Further optimizations would be possible if the execution of interactions would be merged into the reduction system. Similar to the special treatment of applications with statically known operator expressions (instruction *beta*), the construction of intermediate interaction descriptions could be avoided wherever it is possible to predict that the interactions will be executed immediately after construction.

$$
\begin{array}{lrl}
inter\downarrow\ (N\ >\!\!>\!\!=\ M) & C\ S = & inter\downarrow\ N\ ([]\ >\!\!>\!\!=\ M) : C\ S \\
inter\downarrow\ M & C\ S = & inter\uparrow\ M\ C\ S \\
inter\uparrow\ (primInter\ parms) & C\ S = & inter\uparrow\ (return\ R)\ C\ S'\ , \\
 & & if\ primInter\ parms\ S = (R, S') \\
inter\uparrow\ (return\ R)\ ([]\ >\!\!>\!\!=\ M) : C\ S = & & inter\downarrow\ (aor\ (M\ R))\ C\ S \\
inter\uparrow\ N\ \ \ ([]\ >\!\!>\!\!=\ M) : C\ S = & & inter\uparrow\ (N\ >\!\!>\!\!=\ M)\ C\ S \\
inter\uparrow\ N & nil\ S = & (N, S)
\end{array}
$$

Figure 7.14: Iterative interaction interpreter

One aspect of the implementation that might not be immediately obvious from this abstract specification relates to the evaluation of dynamically constructed applications. Of course, it is not necessary to call the compiler at runtime (as the call to *aor* might suggest): the code for the interaction result R and that for the continuation M can be statically pre-computed, and the structure of the applications constructed by the interaction interpreter is always the same (a continuation is applied to an interaction result). It suffices to give the interaction interpreter access to the code of a general apply function which applies its first parameter to its second. This code may then be called with a continuation and an interaction result as parameters.

7.4 Interactions for all valid language expressions

The design of interactions, as described in section 5.3, abandons the restriction of input/output-facilities to strings of characters and allows them to be used with any valid expression of our language. This is in line with the principle of data type completeness and leads to a more uniform and thus simpler language design, but places rather high demands on the implementation. Due to the complex memory structures that are used to represent language expressions, the implementation of the simple high-level operations becomes complex, too, because from this internal representation, a representation in the file system has to be generated on output (*store*) and vice versa on input (*retrieve*). These representation conversions and the movement of data involved can be rather simple in principle, but have to take the various layouts of descriptors, heap objects and code segments into account, resulting in a large number of specialized instances of two generic algorithms. Writing such algorithm instances of similar structure into explicit source code for a great number of data types is tedious, error-prone and a maintenance nightmare, and should therefore be avoided whenever this is possible.

To this end, the data structures could be given a uniform layout to allow one algorithm to cover all cases. As another option, the individual parts of the source code could be generated from formal descriptions of the memory structures and of a generic algorithm. Unfortunately, none of these options is supported by the available implementation, so we tried to reuse available code at least. From an abstract view of the reduction systems, we identified two system components that might offer potential for code reuse. The first is rather obvious: the memory management routines have to deal with all available memory structures. However, these routines can assume to move objects only inside one virtual memory area, and can thus be implemented by code that is not necessarily general enough for our purposes. For instance, the memory organization of the core reduction system never requires descriptors or code segments to be moved, and heap objects can be moved without looking at their internals because all pointers are indirect (the descriptor fields to be modified can be found via a single back-reference stored immediately in front of each heap object). The only memory management routine that needs to traverse (representations of) all nodes of a graph is the deallocator (reference counts of sub-expressions need to be decremented and, if they drop to zero, the memory segments for the representations of the sub-expressions need to be deallocated, too), and even this routine does not touch the code area.

The second alternative arises from the non-sequential variants of the reduction system described here, especially from distributed memory implementations. In order to establish processes on another processor, graph representations need to be copied to the local memory of this other processor, a task that, in principle, requires a complete graph traversal routine (there are no real shortcuts

if the local memories are truly separated). In practice, we found only one major deviation from a general traversal mechanism in the routines: the program graph is divided into a static part (the original program including the abstract machine code) and a dynamic part (graph structures constructed at runtime), and the static graph is broadcasted once to all processors before the distributed execution starts. Afterwards, only nodes in the dynamic part of the graph are transmitted between processor memories because read-only copies of the static graph are available in each local memory (under the same set of addresses) and can be shared between the nodes of the distributed dynamic graph. Apart from this optimization, no specific assumptions are made in the implementation of the graph send and receive routines, and it was possible to adapt the code for our purpose. Several modifications were necessary, mainly related to assumptions about properties of the static graph that are not valid in our application, but basically these modifications 'completed' the available code, i.e., made it less dependent on specific working conditions. The original send and receive routines fall out as a special (optimized) case of the completed algorithm and all the memory-structure-specific code is shared between send/receive and store/retrieve.

Formal specifications of the algorithms to store and retrieve graph structures are given in figures 7.17, 7.18 and 7.19. These algorithms necessarily have to be presented at a very low level of abstraction as they are concerned with the representations of graph structures in terms of descriptors, heap objects and code vectors. Figure 7.15 summarizes the relevant abbreviations used in the description and figure 7.16 lists the data tags which are used in the external, linearized representations of program graphs. For simplicity, we assume that descriptor contents and code vector elements can be treated uniformly, and that the target area of each pointer can be identified from looking at the pointer. In practice, descriptor-type- and instruction-specific routines have to be used, a constraint which adds much to code complexity but almost nothing to the concepts involved.

The basic idea is to do a pre-order traversal of a program graph and to generate a linear representation of the structures found. The main problems are to preserve the sharing of sub-graphs while traversing each shared sub-graph only once, and to keep the pointer abstractions intact while moving memory structures into a different address space. Another problem is that code for functions is not stored on a per function basis but on a per program basis, i.e., in the core system, there is exactly one code vector for one program. Such a code vector contains the code fragments for all functions in sequence, and function calls use direct pointers into the code vector wherever possible. Therefore, both descriptors and code vectors can be shared directly whereas for every heap object there is exactly one descriptor which has a pointer to the object. As a consequence, both store and retrieve keep indexes of descriptors and code vectors that have been transmitted. Each descriptor is transmitted exactly once – further pointers to it are transmitted as indices into the list of known descriptors. Similarly, each code

$const$	: simple constant
ptd	: pointer to descriptor
pth	: pointer to heap object
obj_{pth}	: heap object pointed to by pth
ptc	: pointer into code vector
$head_{ptd}$	: header of descriptor pointed to by ptd
$desc_{ptd}$	: contents of descriptor pointed to by ptd
$instr$	: instruction
soc	: start address of code vector
S	: stack of descriptors to be stored or retrieved
$I_D, (D_i)_{i \in I}$	: index of (local addresses of) transmitted descriptors
$I_{CV}, (soc_i)_{i \in I}$	: index of (local start addresses of) transmitted code vectors
$C, (instr_i)_{i \in I}$	: sequence of local code vectors, separated by $\mid$
max_I	: greatest index in index set I
new_I	: short form of $(max_I + 1)$
$M\{j := x\}$	: short form of $((x_i)_{i \in I, i \neq j}\ (x)_j)$ if $M = (x_i)_{i \in I}$
$M, (M_a)_{a \in A}$	: local memory, consisting of separate areas $(A = D \cup H \cup C)$ for descriptors (D), heap objects (H), and code vectors (C)

Figure 7.15: Parameter names and constructs used in store or retrieve

$C\ const$	: simple constant
$I\ i$	: index of a previously transmitted descriptor
PTD	: placeholder for pointer to descriptor
$DESC\ head$	: head of new descriptor, contents follow
HO	: heap object, contents follow
CO	: code vector, contents follow
$\mid$	: separator
$PTC\ j, p$	: pointer to code, relative to start of code vector j

Figure 7.16: Data tags in the linearized representation

$$
\begin{array}{l}
store\ nil\ I_D\ I_{CV}\ C \\
\quad = \quad nil \\
store\ (const : S)\ I_D\ I_{CV}\ C \\
\quad = \quad (C\ const) : (store\ S\ I_D\ I_{CV}\ C) \\
store\ (ptd : S)\ (D_j)_{j \in J}\ I_{CV}\ C \\
\quad = \quad (I\ i) : (store\ S\ (D_j)_{j \in J}\ I_{CV}\ C) \qquad\qquad , \ if\ (ptd = D_i) \\
store\ (ptd : S)\ I_D\ I_{CV}\ C \\
\quad = \quad (DESC\ head_{ptd}) : (st\_Desc\ desc_{ptd}\ S\ (I_D\ ptd)\ I_{CV}\ C)\ , \ if\ (ptd \notin I_D) \\
\\
st\_Desc\ nil\ S\ I_D\ I_{CV}\ C \\
\quad = \quad | : (store\ S\ I_D\ I_{CV}\ C) \\
st\_Desc\ (const : t)\ S\ I_D\ I_{CV}\ C \\
\quad = \quad (C\ const) : (st\_Desc\ t\ S\ I_D\ I_{CV}\ C) \\
st\_Desc\ (ptd : t)\ S\ I_D\ I_{CV}\ C \\
\quad = \quad PTD : (st\_Desc\ t\ (ptd : S)\ I_D\ I_{CV}\ C) \\
st\_Desc\ (pth : t)\ S\ I_D\ I_{CV}\ C \\
\quad = \quad HO : (st\_HeapObj\ t\ obj_{pth}\ S\ I_D\ I_{CV}\ C) \\
st\_Desc\ (ptc : t)\ S\ I_D\ I_{CV}\ C \\
\quad = \quad st\_Code\ t\ ptc\ (startofCode\ ptc\ C)\ S\ I_D\ I_{CV}\ C \\
\\
st\_HeapObj\ d\ nil\ S\ I_D\ I_{CV}\ C \\
\quad = \quad | : (st\_Desc\ d\ S\ I_D\ I_{CV}\ C) \\
st\_HeapObj\ d\ (const : t)\ S\ I_D\ I_{CV}\ C \\
\quad = \quad (C\ const) : (st\_HeapObj\ d\ t\ S\ I_D\ I_{CV}\ C) \\
st\_HeapObj\ d\ (ptd : t)\ S\ I_D\ I_{CV}\ C \\
\quad = \quad PTD : (st\_HeapObj\ d\ t\ (ptd : S)\ I_D\ I_{CV}\ C)
\end{array}
$$

Figure 7.17: Storing graph structures (part I)

$$
\begin{array}{l}
startofCode\ ptc\ (instr_i)_{i\in I} \\
\quad = ptc \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \ if\ instr_{ptc} = | \\
startofCode\ ptc\ (instr_i)_{i\in I} \\
\quad = startofCode\ (ptc \Leftrightarrow 1)\ (instr_i)_{i\in I} \qquad\qquad\qquad\qquad , \ if\ instr_{ptc} \neq | \\
\\
st\_Code\ d\ ptc\ soc\ S\ I_D\ (soc_i)_{i\in I}\ C \\
\quad = PTC\ j, (ptc \Leftrightarrow soc) : (st\_Desc\ d\ S\ I_D\ (soc_i)_{i\in I}\ C) \qquad , \ if\ soc_j = soc \\
st\_Code\ d\ ptc\ soc\ S\ I_D\ (soc_i)_{i\in I}\ C \\
\quad = CO : \\
\qquad (st\_CodeVec\ d\ new_I\ ptc\ (soc+1)\ S\ I_D\ ((soc_i)_{i\in I}\ soc)\ C), \ if\ soc \notin I_{CV} \\
\\
st\_CodeVec\ d\ j\ p\ c\ S\ I_D\ I_{CV}\ (instr_i)_{i\in I} \\
\quad = | : \\
\qquad (st\_Code\ d\ p\ soc_j\ S\ I_D\ I_{CV}\ (instr_i)_{i\in I}) \qquad\qquad , \ if\ instr_c = | \\
st\_CodeVec\ d\ j\ p\ c\ S\ I_D\ I_{CV}\ (instr_i)_{i\in I} \\
\quad = PTD : \\
\qquad (st\_CodeVec\ d\ j\ p\ (c+1)\ (ptd : S)\ I_D\ I_{CV}\ (instr_i)_{i\in I})\ , \ if\ instr_c = ptd \\
st\_CodeVec\ d\ j\ p\ c\ S\ I_D\ (soc_i)_{i\in I}\ (instr_i)_{i\in I} \\
\quad = PTC\ j, (ptc \Leftrightarrow soc_j) : \\
\qquad (st\_CodeVec\ d\ j\ p\ (c+1)\ S\ I_D\ (soc_i)_{i\in I}\ (instr_i)_{i\in I}) \quad , \ if\ instr_c = ptc \\
st\_CodeVec\ d\ j\ p\ c\ S\ I_D\ I_{CV}\ (instr_i)_{i\in I} \\
\quad = C\ instr_c : \\
\qquad (st\_CodeVec\ d\ j\ p\ (c+1)\ S\ I_D\ I_{CV}\ (instr_i)_{i\in I}) \qquad , \ if\ constant\ instr_c
\end{array}
$$

Figure 7.18: Storing graph structures (continued)

$$
\begin{array}{l}
retrieve\ nil\ S\ I_D\ I_C\ M \\
\quad = \ < S\ I_D\ I_C\ M > \\
retrieve\ ((C\ const) : t)\ (a : S)\ I_D\ I_C\ M \\
\quad = \ retrieve\ t\ S\ I_D\ I_C\ M\{a := const\} \\
retrieve\ ((I\ i) : t)\ (a : S)\ I_D\ I_C\ M \\
\quad = \ retrieve\ t\ S\ I_D\ I_C\ M\{a := D_i\} \\
retrieve\ ((DESC\ head) : t)\ S\ I_D\ I_C\ (M_a)_{a \in D \cup H \cup C} \\
\quad = \ ret\_Desc\ (new_D + 1)\ t\ S\ (I_D\ new_D)\ I_C\ M\{new_D := head\} \\
\\
ret\_Desc\ a\ (| : t)\ S\ I_D\ I_C\ M \\
\quad = \ retrieve\ t\ S\ I_D\ I_C\ M \\
ret\_Desc\ a\ ((C\ const) : t)\ S\ I_D\ I_C\ M \\
\quad = \ ret\_Desc\ (a + 1)\ t\ S\ I_D\ I_C\ M\{a := const\} \\
ret\_Desc\ a\ (PTD : t)\ S\ I_D\ I_C\ M \\
\quad = \ ret\_Desc\ (a + 1)\ t\ (a : S)\ I_D\ I_C\ M \\
ret\_Desc\ a\ (HO : t)\ S\ I_D\ I_C\ (M_a)_{a \in D \cup H \cup C} \\
\quad = \ ret\_HeapObj\ (a + 1)\ new_H\ t\ S\ I_D\ I_C\ M\{a := new_H\} \\
ret\_Desc\ a\ (PTC\ j, p : t)\ S\ I_D\ (soc_i)_{i \in I}\ M \\
\quad = \ ret\_Desc\ (a + 1)\ t\ S\ I_D\ (soc_i)_{i \in I}\ M\{a := (p + soc_j)\} \\
ret\_Desc\ a\ (CO : t)\ S\ I_D\ I_C\ (M_a)_{a \in D \cup H \cup C} \\
\quad = \ ret\_Code\ a\ t\ new_C\ S\ I_D\ (I_C\ new_C)M \\
\\
ret\_HeapObj\ d\ a\ (| : t)\ S\ I_D\ I_C\ M \\
\quad = \ ret\_Desc\ d\ t\ S\ I_D\ I_C\ M \\
ret\_HeapObj\ d\ a\ ((C\ const) : t)\ S\ I_D\ I_C\ M \\
\quad = \ ret\_HeapObj\ d\ (a + 1)\ t\ S\ I_D\ I_C\ M\{a := const\} \\
ret\_HeapObj\ d\ a\ (PTD : t)\ S\ I_D\ I_C\ M \\
\quad = \ ret\_HeapObj\ d\ (a + 1)\ t\ (a : S)\ I_D\ I_C\ M \\
\\
ret\_Code\ a\ (| : t)\ c\ S\ I_D\ I_C\ M \\
\quad = \ ret\_Desc\ a\ t\ S\ I_D\ I_C\ M \\
ret\_Code\ a\ (PTD : t)\ c\ S\ I_D\ I_C\ M \\
\quad = \ ret\_Code\ a\ t\ (c + 1)\ (c : S)\ I_D\ I_C\ M \\
ret\_Code\ a\ (PTC\ j, p : t)\ c\ S\ I_D\ (soc_i)_{i \in I}\ M \\
\quad = \ ret\_Code\ a\ t\ (c + 1)\ S\ I_D\ (soc_i)_{i \in I}\ M\{c := (p + soc_j)\} \\
ret\_Code\ a\ (C\ instr : t)\ c\ S\ I_D\ I_C\ M \\
\quad = \ ret\_Code\ a\ t\ (c + 1)\ S\ I_D\ I_C\ M\{c := instr\}
\end{array}
$$

Figure 7.19: Retrieving graph structures

fragment is transmitted exactly once, but whenever a pointer into a code vector is encountered during the graph traversal, the whole code vector is transmitted at once to preserve the memory layout that is expected by the runtime system.

The algorithm *store* generates a linear representation of a program graph, using one auxiliary algorithm for each kind of memory structure. One of the available stacks is used to keep pointers to descriptors encountered while traversing a memory segment, and memory elements are appropriately tagged for transmission. We assume here that memory elements can be uniformly identified, whereas in practice the context (position in descriptor, kind of descriptor, kind of instruction, parameter position in instruction) needs to be taken into account to get this information. All non-pointer constants can safely be transmitted without modification (C *const*). Pointers to descriptors are either known, in which case only their index is transmitted (I i) or are encountered for the first time. In the latter case their address is added to the descriptor index and their contents are transmitted using *st_Desc* after first sending the general header (basically indicating the kind of graph node represented). Note that the reference count is not transmitted because a complete copy of the graph representation is created. In a descriptor, there may be some constant information, pointers to other descriptors, to heap objects or into a code vector. Pointers to descriptors are saved on the stack until the traversal of the current descriptor is complete, only a placeholder (PTD) is sent so that space for the pointer can be reserved on retrieval. *st_HeapObj* is a simple extension of *st_Desc* to heap objects, where neither pointers to the heap nor to the code area can occur. Handling pointers to code is a bit more involved: first, the pointer is used to locate the start address of the code vector which is used for identification. If this code vector has already been sent, its index is transmitted, together with the pointer to code *relative to the start of its code vector* (PTC $j, (ptc \Leftrightarrow soc)$). Otherwise, the code vector is sent out first and its start address is added to the index of code vectors. During code traversals, both pointers to descriptors and pointers to code can be encountered as instruction parameters and have to be treated correctly. Instructions and other instruction parameters are treated as constants.

The algorithm *retrieve* reconstructs a program graph from its linear representation, basically following *store* in its structure. The stack is used here to save the addresses where pointers to descriptors have to be inserted later (PTD). Initially, it holds the address where the root of the program graph shall be placed. Constants are simply placed in memory at pre-determined locations, indexes of known descriptors are translated into pointers to local addresses unsing the index of descriptors, and descriptors are placed in the local descriptor area. *ret_Desc* stores constants in memory, saves addresses for pointers to descriptors on the stack, and allocates local memory for heap objects and code vectors. It also reconverts relative pointers to code to absolute addresses using the code vector index to get the start address of the code vector in local memory. *ret_HeapObj* is again a simple extension of *ret_Desc* to another area of memory, but this time,

this also holds for the treatment of code vectors (ret_Code).

7.5 Comments on the implementation

All implementation extensions specified in this chapter have been integrated into the existing implementation of the core reduction system. We have refrained from listing and describing the C code here or in an appendix to this thesis for two reasons: such a low-level representation of the implementation would have added a flood of details to be described without adding anything new conceptually, and it would have been difficult to disentangle our contributions from the existing code at this level (mainly due to the complexity of interactions and interfaces between the parts of the implementation at the coding level). Instead, we have given abstract specifications of the existing system and of our extensions in sufficient detail to allow for a straightforward translation of the specifications into C code. Of course, our abridged description of the implementation of the core language does not reflect the full complexity of the complete reduction system, but it suffices to describe the implementation of our extensions and their relationship to the core system. The form of presentation results from an attempt to describe a full view of the implementation, trying to directly relate the language definition to the specification of the implementation by focusing on the implementation decisions involved. Therefore, the presentation of the whole system differs from previous accounts, e.g., by factoring an algorithm that implements a reduction strategy into its static and dynamic parts instead of just postulating an abstract stack machine and defining a compiler from the programming language into abstract machine code. We have tried to document the decisions that lead from our language definition to an implementation very close in spirit to the existing one. As is explained below, a documentation of this kind for the complete system would not only provide rationales for the form and existence of all parts of the implementation, it would also allow to revise implementation decisions in response to modifications of the language definitions.

Our attempts to adapt the existing implementation to accommodate our language extensions unveiled a number of problems below the level of the previously existing implementation specifications. While the general implementation design should be quite flexible with respect to extensions, the translation of this design into C-code has several properties that hinder us to explore the full potential of the implementation design. To give but one example, program evaluation has been factored into three major phases: a preprocessing phase prepares the program representation for more efficient execution during the real processing phase, and a postprocessing phase undoes all representation changes to produce a high-level representation of the evaluation result. In brief, each preprocessing step should be reversible, and its inverse is used as a postprocessing step. If no reductions or interactions are performed during processing, the combination of

preprocessing, processing and postprocessing should be the identity. We hoped to make use of this property by invoking the compiler on the results of the processing phase before writing them to the file system. However, the implementation code is "optimized" in several ways to depend on the sequence of phases. In other words, a program representation *before* a preprocessing step is *not* equivalent to the representation *after* the corresponding postprocessing step. To get an equivalent representation, the upper part of the usage cycle (cf. figure 7.9) has to be followed or at least simulated. This "feature" of the implementation is not part of the specification, but has somehow found its way into the code. Unfortunately, the different concrete representations at the coding level are mapped to equivalent abstract representations at the specification level, and so the code can still claim to conform to the implementation specification[3].

Given that formal specifications exist both for the language and for the implementation, the basic problem is that the gaps between the different specification and code layers are too large. Moreover, the layers are only 'upwards-related', i.e., it is possible to check whether the implementation specification satisfies the language specification, or whether the implementation code satisfies the implementation specification, but the relation is always upwards in the abstraction hierarchy. The higher-level specification is assumed to be fixed, and a lower-level specification is then 'invented' to implement the abstract specification. The lower levels of specification just happen to be there, and even if they conform to the higher levels, there is no easy way to relate the language definition to the implementation code. Therefore, every major modification or extension of the language requires a sophisticated reverse engineering process with the goal to relate parts of the implementation code to parts of the language definition (to avoid the invention of a completely new implementation). Starting with such a reversed view of the complete system, it may then be possible to identify large parts of the existing implementation as reusable for the new language definition, or to relate modifications in the definition layer to modifications in the coding layer.

We have come to the conclusion that the situation could be improved drastically if (a) the gaps between specification layers would be smaller (in particular, there should be no substantial implementation decisions between the final implementation specification and the coding level), and if (b) there existed a description of how these specification layers are related 'downwards' (cf. figure 7.20). Basically, we seek to replace the very course-grained and non-deterministic "*conforms to*"-relation by a more fine-grained equivalence transformation. Of course, a concrete specification cannot be directly equivalent to an abstract one – some details have been added. On the left-hand side of figure 7.20, this could be accounted

[3]The specification does not explicitly require the mentioned equivalence of representations at all levels of the usage cycle, but only at the user interface level, where the differences in the internal representations are not observable.

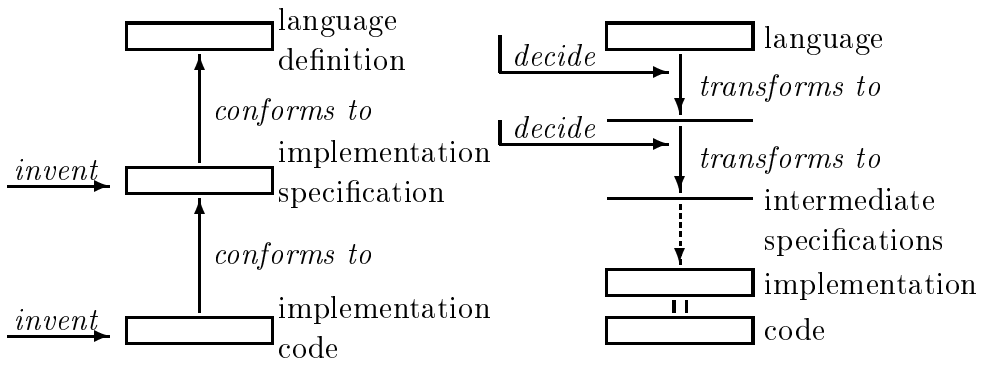

Figure 7.20: Implementing specifications – invent & test versus transformation

for by introducing an observation relation that forgets or hides the additional details, so that "L *conforms to* H" could be read as "the observed behavior of L is equivalent to that of H". While this may be adequate for someone mainly concerned with the high-level views, it is simply not acceptable for those who need a complete view of the system: the "unobservable" details are exactly what matters about the implementation! Therefore, we propose another approach to establish an equivalence between specifications of different abstraction levels: instead of forgetting information about the more concrete specifications, we add information to the more abstract ones. The additional information turns out to represent the implementation decisions made, and thus exactly what distinguishes the one chosen concrete specification from the many other ones that would also conform to the abstract specification.

The informal derivation of an implementation in the first section of the present chapter may be seen as a first step in this direction. One of the expected advantages of such a description is that every major implementation decision has to be justified and made explicit in terms of a transformation of the specification. It is then easy to identify and eliminate those transformations that are invalidated by a modification of the language definition, and a first variant of the modified implementation can be generated by replaying the remaining transformation steps starting from the modified language definition. Ideally, the existing implementation code can be adapted by incremental modifications which do not have to be guessed but can be derived from the modifications to the higher specification levels. Such a derivation approach to implementation nicely expresses the view that the programming paradigm to be supported determines the implementation architecture (and not vice versa) and thereby provides better support for extensions or modifications of the language definition. The derivation can even be given a formal basis, e.g., if specifications on all layers of abstraction are uniformly represented as systems of object transformations.

Chapter 8

Related and Further Work

As has been demonstrated in the previous chapters, we have have fulfilled the original task specification to extend the existing functional language and its implementation with facilities for modular programming and input/output. We have also met our additional requirements for a simple language design with general and completely integrated components. This work has been shaped by an unusually large number of connections to other work, partly due to the need to build on an existing core language and its implementation and partly due to the integrative nature of our task, involving several areas of high research interest. In addition to results from the individual research areas, our quest for a simple and general language design has also allowed us to build on a large body of general language design work. Most of these connections have been mentioned in earlier chapters of this thesis and are not repeated here. Instead, we focus on general options for further work and elaborate on the connections to research on type systems and persistence. In particular, we explain why we have purposely excluded all aspects of type systems from our work, and thus an area of research with important relations to our topics, and we argue that our generalization of input/output to all valid language expressions opens new possibilities to combine the research areas of functional programming and persistent systems.

8.1 Options for further work

There are various possible directions for further work, related to the design of functional languages, framework design for modular programming, or to the classification of abstractions used for modular programming. One of these directions is related to a topic we have deliberately left open in our description, namely the details of the functional core language in our design. Of course, we have added our extensions to a complete language and its implementation, but an evaluation or possible redesign of these given components was beyond the scope of our work. The questions arise to what extent the core language complies with the language

design principles we have used throughout this thesis, and how our abstract language design can be refined to encompass all details of a full programming language.

Refining the language design

We have described an abstract framework for extending purely functional languages with facilities for interactions and modular programming, leaving unspecified the details of the particular core language we have been working on. We have only required that it supports a full λ-calculus, which is necessary to define suitable abstractions over functions, frames and interactions and thus to fully integrate the parts of our language framework. The obvious next step is the refinement of our design to address the issues of the functional core, simplifying its advanced features according to our design principles and integrating the primitive operations and interactions into a modular and extensible language design. For us, this corresponds to a redesign of the given core language and implementation, but there is also a second possibility to complete our abstract design into a fully specified language: since we have purposely avoided the specification of details that are not relevant to our topics, one can start from any given functional language that meets our requirements[1] and use our abstract design to complete this particular language with respect to interactions and modular programming. Most current functional languages do already provide some scheme for character-based input/output and the extension to support interactions for any valid language expression basically seems to be an implementation issue. The most difficult tasks in translating our design to other functional languages are related to restrictions imposed by the static type systems that are in widespread use today. A summary of the typing problems to be expected is given in section 8.2.

A preliminary evaluation of the given reduction language with respect to the design principles used in this thesis revealed several aspects that need further investigation. On the positive side, both the language and its implementations are dedicated to support a full λ-calculus, the only exception being a bias towards applicative-order reduction strategies in most implementations. We can hardly overemphasize the importance of this feature, not only because the λ-calculus provides the means for abstraction in our design and thus the glue for a full integration of its parts. Without the execution model of high-level program transformations, e.g., it may still be possible to use λ-abstraction for the static description of parameterization of program components, but programmers could never see the results of computations that involve program components. And the idea of first class program components that can both be stored in the file system and retrieved from there at runtime would soon cease to make sense if it

[1] Surprisingly, many modern functional languages and implementations do *not* support a full λ-calculus, even though most claim to be based on one.

would not be possible to generate high-level representations of the program components that currently exist in the long-term storage of the file system, including high-level representations of the functions included in these components. If the execution model of program transformation would only be mapped to low-level state transformations (with no high-level representation of results unless explicitly generated as program output), programmers would always have to go back to the original program sources and would need to replay mentally all computations on the long-term store that might have affected the components since their first storage. In effect, the advantages of our language design would come to nothing under such conditions.

On the negative side, some of the advanced features of the core language would be better defined in terms of a few elementary constructs. The most prominent example of this are the very complex pattern matching facilities, which have developed into a kind of sub-language for case distinctions and structure decompositions. The problem here is not the expressiveness of pattern matching, but the hard-coding of pattern matching facilities in the language definition. Not only does this add to the complexity of the language and its implementation, it is also a serious obstacle to extensibility. Whenever the language is extended with a new kind of data structure, the pattern matching facilities have to be extended, too (the same holds, of course, for modifications of the language and its implementations). This additional burden does not even pay of for programmers, as the fixed pattern matching constructs cannot be extended in any way, e.g., to work with user defined data abstractions. Instead, every change to the representation of a data abstraction is likely to break applications of pattern matching at various places in the program, which runs counter to the basic ideas of modular programming (this phenomenon is well known from other languages that support pattern matching and data abstraction [Wad87]). On the other hand, both pattern matching and case distinctions are important language features and it would be a worthwhile area of research to work out the elementary language constructs behind these currently rather complex features and to recombine them in a way that does not suffer from the aforementioned problems. Some of the preliminary work in this area suggests to explore the relation between extensible records, variants, fold operations and the current `case`-constructs based on pattern matching [MJ95, GJ96, Dam96].

Another problem of the functional core language with respect to modularity and language extensibility is the organization of δ-rules (definitions of primitive operations), which form a completely unstructured set at the language definition level. The implementation and documentation are much more structured in this case, organizing implementations and descriptions of primitive operations in different modules according to the kinds of data objects they operate on. This organization is, however, not reflected at the language level, and the language presents itself to programmers on an all or nothing basis as far as built-in operations are concerned. It seems advisable to (a) review the existing primitive

operations and to remove some of them in favor of predefined functions, (b) move as much as possible of the basic operation framework from the language definition to areas that can be extended by the user, (c) separate the primitive and predefined operations into small groups, e.g., using the module facilities just introduced. In defining the primitive interactions, we have followed the existing practice for primitive operations, so the same arguments apply to this part of the design, even more so as the kinds of interactions that can be supported depend on design decisions in the operating system, thus out of our reach. Probably, environments should be modeled as abstract data types providing certain operations to be called from the programming language via the general framework for interactions. This way, the file system interface could easily be extended or replaced by a database interface or by an interface to any other component of the runtime environment in which the program is to be run. A minimal interface needs to be defined along the lines of section 5.3 and language mechanisms for interface extension need to be provided. If the environment consists of many different kinds of objects, a modular style of per object interfaces would be necessary, too.

This is essentially the same search for elementary language features that led us to the proposed framework of functions, frames and interactions on a larger scale, only pursued further into the parts of this framework. The problems of the given core language and implementation are a direct consequence of the so far missing support for modularity and the results of our work are thus one prerequisite for their solution. However, even though a module facility is available now, there are still several possible strategies to apply this facility to the problems mentioned above, and the decision in favor of one of the possibilities is not an easy one. The problems affect a fundamental level of the language and any decision would therefore have far-reaching consequences. In any case, a major redesign of the core language and its implementations would be required, making the whole problem area a topic for further work.

Review of the primitive interactions

Adding imperative features to a purely functional language, even in a safe way, bears the risk of pushing the level of abstraction on which programs are written back to a state where programmers have to deal with cumbersome low-level details again. There are two kinds of answer to this concern: on the positive side, the integration of the new features has been done in a way that carefully avoids any negative effects on the functional subset of the new language and even extends most of its properties to the full language, which is thus no longer purely functional but still a pure functional language. This means that the full expressiveness of the functional language is available to construct imperative functional programs on the same level of abstraction used to construct functional programs. The second, less positive answer is that the language level is not only affected by the means for program construction but also by the available primitives. In

the current design, the primitive interactions are based on the input/output-operations available in a UNIX/C environment [Tim96], and so it was clear from the beginning that these operations would probably be too much concerned with low-level details. This leads to rather fine-grained interactions and thus to a granularity mismatch between the interaction-based and the reduction-based parts of the language. While the granularity of interactions is just fine for imperative languages, where every computation step involves an interaction with a global store, it may not fit well into a reduction language, where programs are usually much less concerned with the details of how computations are performed.

In spite of this, it was decided to base the first implementation on these operations for two reasons: (a) these are the operations our users were thinking of when they complained about missing facilities for explicit input/output, and (b) we did not have any experience with the combination of low-level input/output-operations and a high-level programming language. Without knowing the problems, it would have been a bad idea to design a new set of operations out of thin air. Furthermore, the facilities of the functional language enable users to build their own control abstractions on top of the primitive interactions as long as the latter provide the necessary functionality. Nevertheless, it will be necessary to evaluate and possibly redesign the set of input/output-operations as soon as useful abstractions have been established and sufficient practical experience is available.

Framework design for modular programming

Once the abstract design framework has been instantiated with a concrete functional core language, the next necessary path of research is the identification of commonly used abstractions. In chapter 6, we have shown how various well-known language constructs for modular programming can be modeled in our language design, based on the idea that all these different constructs are just instances of a general scheme of abstraction. We have also argued that none of these constructs should be introduced as a language primitive since none of these constructs is really fundamental. Rather, some of these constructs should be provided as a second level of abstraction on top of the fundamental constructs of our design. Our rationale for this decision was to provide programmers both with a set of useful predefined abstractions and with the means to adapt the implementations of these abstractions to their needs or to define new domain-specific abstractions. However, for a practical language environment, useful abstractions have to be identified and implemented to provide a library of constructs for modular programming. Guidelines for the choice of abstractions for particular problems, for the use and composition of individual abstractions, and for the design of new abstractions have to be developed. In brief, frameworks for modular programming need to be developed in which the different kinds of software components can be used effectively in practice.

Essentially, this work would be an elaboration of the ideas presented in chapter 6, where we have developed the idea to formalize the process of abstraction in section 6.4 to provide a basis for the modeling techniques used in our examples. The description given there is still in a simple form, but we were able to relate various informal uses of the term abstraction to a few formal concepts. Using these concepts, we have attempted to develop a simple classification of constructs for modular programming which could all be identified as abstractions emphasizing different aspects of programs or different phases of the abstraction process. It would be interesting to further investigate this classification to develop a better understanding of the pros and cons of the various known abstractions, not only to prepare for the development of useful abstractions for future functional languages, but also to provide a formal basis for the discussion of existing languages with explicit support for modular programming.

Generalized program transformations

The user interface of the reduction system [Klu94] to which we added our extensions features a syntax-oriented editor for inspection and manipulation of programs, be they user-specified or (intermediate) results of reductions. This editor provides static support for partial views of programs, i.e., programmers can freely move their focus of attention to any part of the program, temporarily ignoring the program context in which the part is embedded (they also have some control over the style of abbreviation of sub-expressions that is used if the selected program part is to large to fit on the screen). However, as soon as a reduction sequence is initiated, abstractions inside the current focus of attention may be substituted by their definitions. This is not so much a consequence of the execution model of program transformations as a consequence of the very plain interpretation of this model: independent of the high-level abstractions described by a program, execution proceeds simply as a step-by-step, one-rule-at-a-time process. Not only does this cause an exponential growth in program size during stepwise transformation (function definitions are substituted into all sub-expressions to be available there if the focus of attention is shifted between steps), unfolding all abstractions also burdens programmers with details they may not at all be interested in. Essentially, the unfolding of function definitions during reduction causes all previously established levels of abstraction to collapse into one – the system does not distinguish between reductions in different levels of abstractions. Therefore, abstraction is supported only statically by the current systems, it is not preserved during reduction. On closer investigation, it turns out that these inconveniences at the language level also have very unfortunate consequences for implementation efficiency.

Usually, programmers establish abstractions via function definitions when they want to distinguish between a level of function definition and a level of function application. On the level of function definition, they may be interested

in all details of a typical function application. For instance, they might want to reduce applications of their function definitions on a step-by-step basis to verify their correctness, and of course the reduction system supports this. On the level of function application, however, programmers are no longer interested in the details of the definition or even in the exact sequence of reduction steps from a function application to a result. But this is exactly what the current system has to provide for: even if programmers ask for an unbounded number of reduction steps to be performed, indicating that they are only interested in a final result, the implementation has to guarantee that the program is executed on the basis of a step-by-step reduction. One of the unfortunate consequences is that some common optimizing program transformations cannot be used because they would cause the program execution to deviate from the usual reduction sequence.

In brief, the programmer-chosen levels of abstraction are not respected by the current implementation, and programmers even have to pay for this unwanted feature by an implementation that cannot fully optimize programs. Even on the system implementation level, the effects are rather negative because implementors have to take care to comply to the pre-defined reduction sequences instead of implementing more optimal and sometimes even simpler shortcuts. In view of these negative impacts on all levels of the reduction system, an obvious improvement would be a feature that allowed programmers to mark those function definitions below their current level of abstraction (in contrast to the focus of attention, the current level of abstraction is a non-local aspect of programs). Individual reduction steps in 'hidden' function definitions should then simply be not observable at the user level, and the implementation would be free to fully optimize these intermediate reduction sequences. Just as function names are used as textual abbreviations for function definitions, the single high-level step from a function application to a function result should be used as an abbreviation for the sequence of low-level reduction steps involved.

Unfortunately, neither the precise definition of reductions on different levels of abstraction nor the details of an implementation are as simple as it might seem. For instance, the final boundaries of high-level reduction steps need to be defined, too, and high-level applications need to be reconstructed if the corresponding low-level reduction sequences do not run to completion, e.g., if the user specifies an upper limit on the number of reduction steps that is too low. Furthermore, if an abstraction involves interactions, we face all the problems of *transactions*, known from the database world, only in a more complex setting (cf. also [NW92]). Therefore, an investigation of this interesting high-level view of the program transformations in the language definitions has to be postponed as a topic for further work. Another idea in this class is to explore the potential of using transformation rules of the language as a basis for program structure modifications: so far, the rules have been used only for program evaluation or for formal program transformations that involve the functionality of programs (derivation of programs from specifications, proof of equivalence of programs with different

reduction properties). The major topic of programming-in-the-large, however, is not the functionality, but the structure of programs, and a major problem for programmers is the need to modify the structure of large programs (to alleviate maintenance or to optimize extensibility and reusability). These tasks should be given a formal basis in the transformation rules and they should also be supported by the language implementation.

8.2 Type systems

We have purposely excluded all aspects of type systems from our design. The rationale for this decision was our impression (based on an initial search of the project-relevant literature) that the limited capabilities of current static type systems would not have allowed an unbiased approach to our design tasks. Indeed, it seemed as if the language features we had in mind were particularly suited to highlight the problems of current static type systems. On the one hand, this phenomenon had already stimulated a flood of research on the further development of type systems, so there was no need for us to join this trek. On the other hand, this research had only produced partial solutions to isolated problems (see below), and no unified framework for the proposed type system extensions was available in which our design tasks could have been pursued. Trying to force an extended language design into the constraints of one of the established static type systems was also not felt to be a reasonable option as the design inescapably would have been biased to fit into a type system that was known to be inadequate for our tasks. Instead, we decided to keep the implicitly and dynamically typed nature of the given reduction language and to use this unrestricted framework to get an unbiased view of the design options. We hoped that, ideally, the continuing research on type systems would produce acceptable solutions to the typing problems related to our language design. In any case, it made sense to develop a design for the complete language before looking for a suitable type system.

This approach was very successful in that it allowed us to develop the simple, yet expressive language framework that has been presented in the last three chapters. It also allowed us to disentangle the properties of purely functional languages and static type (inference) systems and to investigate the prospects of purely functional languages in isolation. We have not elaborated on this in chapter 6, but we have come to the conclusion that there is no fundamental discrepancy between functional and object-oriented programming (this follows from our view of abstractions for modular programming as well as from semantic accounts of object-oriented programming languages and the availability of several approaches to state-based computations in functional languages). An immediate consequence of this insight is that the major factor that has delayed the translation of very successful approaches to object-oriented programming in Lisp [DG87] to purely functional languages for more than a decade is the prevailing

assertion that static type inference systems in their current form are not limiting the expressiveness of functional languages in areas of practical relevance. Wrong as it is, this assertion is the rationale for not permitting language constructs and programs that cannot be handled by the type systems attached to current functional languages. Nevertheless, our work would not be complete without at least a summary account of the problems which our language design would pose to the addition of a static type system. We do this here by providing references to some of the ongoing research work on these problems.

Starting with the functional core languages, it is well known that the very first step, from untyped to typed λ-calculus, loses computational completeness: self-applications, an essential prerequisite for the definition of fix-point combinators as terms of the λ-calculus, are not typable in the simply typed λ-calculus. Therefore, it is necessary to add explicit fix-point combinators or other forms of recursive definitions as primitives to the typed calculi if they are to be used as the core of general purpose programming languages (in contrast to their definitions as λ-terms, fix-point combinators themselves are typable in these systems). It is also common to use a polymorphic λ-calculus, because the simply typed λ-calculus does not allow abstraction over expressions of different types, and to avoid explicit type annotations. However, the Hindley-Milner type inference system [Mil78] that forms the core of type inference in current functional languages computes type information only for untyped terms of a predicative polymorphic λ-calculus (no abstraction over expressions of polymorphic type). To allow at least for declarative abstractions over polymorphically typed expressions – and thus the means to define polymorphic functions, the `let`-construct is given a special meaning: `let`-bound variables, in contrast to λ-bound variables, may have a polymorphic type. This is in conflict with the principle of correspondence (no parametric abstraction over polymorphically typed expressions), and it does not allow polymorphic functions to be passed as parameters to other functions – polymorphically typed expressions are not first-class data objects (against the principle of data type completeness).

Furthermore, the type inference problem for polymorphic recursion has been shown to be undecidable [Hen93, KTU93], so that each recursive function can only be instantiated to a single type inside its definition. The same restrictions that do not allow polymorphically typed expressions to be passed as parameters (no local quantifications in type schemes) do also preclude their storage in data structures, so that frames could not sensibly be used to represent modules or objects. In brief, the means for abstraction that form the very core of our design are severely limited in standard static type inference systems. Moving on to the second-order polymorphically typed λ-calculus [Hue90, part II] would allow local quantification of types (and thus polymorphic parameters or structure components) at the expense of fully implicit typing[2]. Mixtures of type annotations and

[2]It seems as if even type-checking in the full calculus is undecidable [Wel96].

type inference can be used in hope to get the best of both worlds [OL96], or local quantification can be restricted to certain contexts (data structures, in particular) to provide the necessary hints for type inference without explicit type annotations in the program (the annotations are abstracted out to the data structure type definitions [Jon97]). So much for the typing problems related to the core language – for a more detailed description and for further references see, e.g., [Mit90].

As mentioned above, each of the proposed language extensions induces further problems for static typing. Frames, for instance, are record-like data structures that combine heterogeneously typed components with dynamic access. These two properties, taken together, stress the capabilities of any static type system as the type of a selected component may depend on the dynamic evaluation of both the selector and the frame. In general, there is no way to determine the type of an expression selected from a frame statically and unambiguously. At best, safe approximations of sets of possible types of selections can be computed, and programs can be guaranteed to be well-typed for every possible type of selections statically (due to dynamic frame modifications, these approximations may end up including all possible types). The actual type of a selection, however, can only be determined at runtime, when the appropriately typed variant of the selection context has to be selected. For similar reasons, statically typed languages usually allow only structures with one of the above mentioned properties (lists and arrays allow dynamic selection, but are homogeneous collections, i.e., all elements of a collection have the same type; tuples may have heterogeneously typed elements, but no selectors of statically unknown type).

As a consequence, support for record-like data structure in current functional languages is restricted in several ways to account for the limitations of their static type inference systems. In Standard ML [MTH90], records are treated similar to tuples[3], i.e., the exact structure of records (the set of slot names and types) has to be known statically for a record operation to be well-typed. There is simply no way to define a general selection operation that takes a field label and a record as parameters and selects the named field from the record (field labels are not even first class values, but are mapped into patterns of a selector function). The same holds for record extension or modification of fields. In Haskell 1.3 [PH96], record-like structures have been introduced by allowing components of data constructors in data type definitions to be labeled using field names. The use of such field names implicitly generates a global selector function of the same name (field names cannot be shared by multiple data types, only by fields of equal type in different constructors of one algebraic data type definition). In general, field names 'do not change the basic nature of an algebraic data type; they are simply a convenient syntax for accessing the components of a data structure by name rather than by position' [HPF97]. They have been introduced for good

[3]More precisely, tuples are defined as records with numeric field labels.

reasons, but are not meant as a general purpose record facility.

The problems of adding records to statically typed languages have been known for about a decade now and some of them have been addressed in the work of several authors. The fact that the typing of flexible record operations poses similar problems as the typing of objects and classes in object-oriented programming languages has been a major motivation for this work (see [GM94] for a collection of papers on these topics). The major issues of the ongoing research work are polymorphic operations on records of different structure (general selector functions, record updates without loss of type information, structural subtyping), record structure modifications (adding, renaming, removing fields, record concatenation), and local type scheme quantification (universal quantification to allow polymorphic functions to be stored inside records, and existential quantification to allow for the typing of operations on abstract data type representations). Other topics are the static inference of the absence or presence of fields in records and the efficient implementation of record operations. In their full (dynamic) flexibility, our frame operations cannot be typed statically, but the limits of static type systems have been extended far enough that a reasonable subset of these operations can be integrated into statically typed languages. Practical experience with implementations of these advanced type systems will be needed to verify whether the remaining limitations really affect our design goals in practice. However, the integration of the partial solutions into a single system and their interaction with other advanced features of modern type systems is still an open research topic (see [GJ96] for a recent attempt [4]; the paper also includes a brief summary of the state of the art and further references).

The generalized input/output-facilities raise typing problems, too, as they allow expressions to be separated from their originating programs or to be introduced into other programs at runtime. Expressions can be stored permanently in the file system and may thus outlive program executions – the factorization of program execution into static and dynamic phases (the very basis of static type-checking) is neither sufficient nor absolute in such a context. Rather, attributes such as static and dynamic have to be interpreted relative to individual programs, which may be evaluated and type-checked in two phases as long as they do not interact with the environment of long-living expressions. For these interactions, however, some dynamic typing is necessary: whenever expressions are stored in the file system, information about their types needs to be attached to them, and whenever expressions are loaded into running programs, their type information needs to be checked for consistency with the types expected by the programs that load them. The interrelations between static and dynamic typing can be captured nicely by introducing a special type `Dynamic`, as proposed in several variants by several authors (we describe here a simplified variant of the

[4] Indeed, this seems to be the first publication to address the integration of the existing solutions and the simplification of the resulting type system.

schemes described in [ACPR94]). Basically, `Dynamic` is a tagged sum of all types, and by injecting an expression into `Dynamic`, it gets tagged with a representation of its actual type (which may be statically inferred). To project an expression from `Dynamic`, a `typecase` construct is used to provide alternative programs for every possible 'real' type of the expression. A runtime type-check compares the representation of the real expression type with the expected types and passes the expression (projected from `Dynamic` to its real type) to the program alternative that expects an expression of this particular type. Each alternative can be type-checked statically, and the interface between static and dynamic type-checking is clearly defined: a runtime representation of a static type is created on injection into `Dynamic`, and a dynamic type-check is performed on projection from `Dynamic`. There are, however, further problems in the details of this proposal and its interaction with other type system features.

Unfortunately, this still does not account for all aspects of typing our extended input/output-scheme: programs that store expressions and those that retrieve expressions from the file system may not even agree on the types they use, indeed, an expression in the file system may outlive all programs that know about the definition of its type. Essentially, the common sequence of static type-checking, compilation and execution no longer adequately describes the whole process of program development, but is reduced to a partial view of the complete system, in which long-living expressions exist in the file system before, during and after all three phases of the sequence, and in which other programs may be working on the same long-term store before, during, and after individual programs proceed through each of these phases. Of course, many of these aspects are relevant even for more conventional language designs, but the extended input/output-system brings these problems into the scope of the language design process (instead of delegating them to programmers). The general consequences of this extended view of language design are being explored in the research area of persistence (cf. section 8.3). For explicitly typed languages, this change of scope means that all of these activities have to be supported by the type systems (contrast this with the untyped or character-based view of the file system in conventional language designs): as a first step, the interface to the file system needs to be typed, e.g., using some variant of dynamic types, and the necessary extensions need to be integrated into the often sophisticated type systems of modern functional languages (cf. [Pil96] for a description of this process for the language Clean). Even basic issues such as type equivalence checking (by name or by structure) need to be reconsidered [CBC+90]. Over time, expressions in the file system may also outlive language and implementation versions, or may need to be adapted to modifications of their original type definitions. These problems have been investigated under the name *system evolution* in the research area of persistent systems [MCC+93, KCMS96].

This concludes our sketch of the first level of typing problems, but the list of problems continues on the next level: so far, we have only discussed how

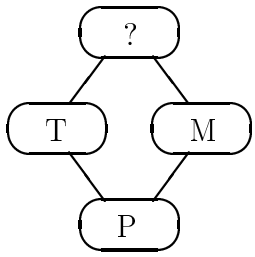

Figure 8.1: Adding types (T) and modules (M) to a programming language P

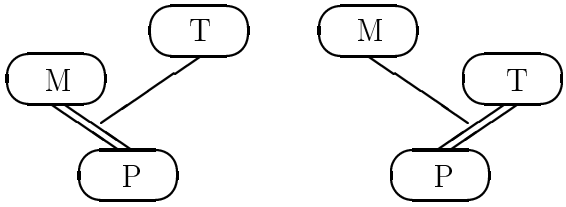

Figure 8.2: Typed modules versus modular types for a language P

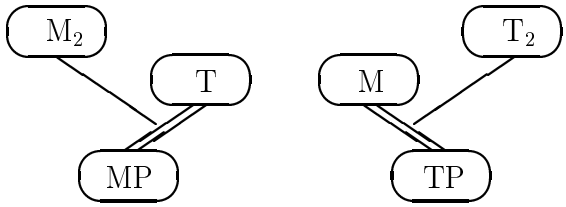

Figure 8.3: The next steps in the hierarchies of meta languages

to type elements of our language design, but if the means to define types are included into the language, the whole design process needs to be reiterated for the new, explicitly typed language. In other words, we need modules of type definitions, storage of type definitions in the file system, abstraction over types, etc. . We sketch only the problems related to support for modular programming in typed languages here, since this has been a very active area of research. First, note that both modules and types are constructions (or abstractions) built over expressions of the programming language, and that there are two alternative ways to proceed when adding both to a language (cf. figure 8.1). Either the constructs for modular programming (M) are introduced before the type system (T) or vice versa, leading to a typed modular language (TMP) or to a modular typed language (MTP). The former allows modules to have types and is thus one prerequisite for modules as first-class values in a typed language, and the latter

allows type definitions to be included into modules (cf. the two sides of figure 8.2). For the typed modular language TMP, an additional level of the module system (M_2) is needed to allow for modular programming with types, and for the modular typed language MTP, an additional level of the type system (T_2) is needed to allow for typed programming with modules (cf. figure 8.3). To avoid the construction of an infinite number of meta levels, the process is usually stopped at this level, either by leaving an open end at the top (for instance, no types for the constructs of M_2) or by trying to build a closed loop (for instance, by merging M and M_2). The obvious questions are whether the process has fix-points, and whether the two lines of development may converge, but these questions are usually not even posed. Instead, the line of development that fits a given language best is pursued as if it was the only possible one.

Examples of this phenomenon can be found by comparing the facilities for modular programming in Haskell [PH96] and Standard ML [MTH90, MT91, HMM86]: Haskell has a rather straightforward module system to organize source code (including function and type definitions) into modules, and the focus of language design has been on the development of a sophisticated type system inside the module structure (matching the right-hand side of figure 8.2). Techniques for modular programming were then developed with no particular support from the module system, using mainly facilities of the type language (type classes in particular). There are, however, proposals to establish further language support for modular programming below the level of the simple module system by providing first-class record structures in the programming language, together with the necessary extensions to the type language [Jon95, Jon96, JJ96]. In other words, Haskell seems to drift towards the situation symbolized in the left-hand side of figure 8.3, with a simple module language M_2 at the top of its hierarchy and a sophisticated second module language below the type language.

Standard ML, on the other hand, has a rather straightforward type system for its core language, and the focus of language design has been on the development of a sophisticated module language on top of the typed core language. While the starting point was similar (right-hand side of figure 8.2), the direction of development was to the outside in this case. This led to the development of the functor and structure language [Mac85], with signatures playing the role of types in T_2 (cf. the right-hand side of figure 8.3). The language of functors and structures forms an own typed first-order functional language and is separated from the core language (so T_2 is built directly on M, not on MTP as shown in the figure). There have been several attempts to bring T and T_2 closer together, viewing them as parts of one type universe (allowing modules with type definitions to be first-class data objects of the core language), or at least permitting higher-order functors (see, e.g., [HL94, Ler94, Ler95, Lil97]). Nevertheless, the current tendency in the ML community is towards keeping the stratified nature of the language. According to Mark Lillibridge [Lil97], this will also be true for the design of the successor language to Standard ML, currently called ML2000.

It would be beyond the scope of this thesis to elaborate on the problems encountered at this level of typed language design (see [Jon96, Lil97] for summary accounts of the field), but note that the problems are entirely on the static typing side, especially on finding reasonable compromises between the flexibilities of dynamic manipulation of modules and the restrictions of static typing. This is even more difficult if the module language becomes 'an organic extension of the underlying polymorphic type system' [Mac92], as it is the case for Standard ML: module manipulation subsumes type manipulation, and a central topic of the work of Lillibridge and Leroy is the wish to maximize the visibility of type manipulations in higher-order module languages while guaranteeing termination of static type elaboration. To solve this apparent conflict, they invented *translucent* (or *manifest*) types for modules with type components: only parts of the internal type information are visible through module boundaries, and by making these boundaries more or less translucent, the type system can statically compute safe approximations of dynamic module manipulations. While the idea looks simple (from hindsight), the technical elaborations are rather complex, and it is no surprise that the technical difficulties left almost no room to scrutinize fundamental design decisions. It has become almost impossible to disentangle the type and module systems in Standard ML without giving up the character of the language, and advances in the module language go hand in hand with advances in the type system. By reflexivity, it follows that advances in the type system (for instance, to accommodate it to the requirements of language constructs for object-oriented programming) should be reflected in the module system to avoid duplication or inconsistencies of features in these two parts of the language.

On the other side of the spectrum, one motivation for the work of Jones [Jon95, Jon96] is the hope to find less complex foundations for first-class modules in statically typed languages. Using Haskell as his starting point, his major problem is to identify and change those aspects of the type system that hinder programmers from using the programming language for modular programming. He tries to separate the issues of modules and types, but runs the risk of duplicating features of the sophisticated type language in the module language. For instance, a standard implementation of type classes implicitly passes *dictionaries* of functions around as evidences for the membership of types in classes [HHJW96, JJM97, Aug93]. Currently, the program transformation that introduces dictionaries as additional parameters to functions may produce programs that could not be written in Haskell itself (the type system would not allow polymorphic functions in dictionaries), but these programs would be admissible in an extended language. Additionally, his proposals have been criticized as being incomplete [Lil97, p. 310], because they do not allow type components in first-class structures. By analogy with the approach used for Standard ML (merging T and T_2 in the right-hand side of figure 8.3), this problem could be solved if it would possible to merge first-class structures and the existing module language (corresponding to M and M_2 in the left-hand side of figure 8.3). This would depend on

further extensions to the type and structure system and would probably lead to problems similar to those encountered in the context of the Standard ML module language.

The stratified language designs described above have some unpleasant properties: just as the two module languages in an extended Haskell would have different characteristics, the two type languages of Standard ML differ in their capabilities, too. Some of these differences were intended (the possibility to abstract over types in T using expressions that have types in T_2), others probably not [Kah93], and the sum of differences can be confusing to programmers, who are forced to decide early in their projects which of the levels to use for modeling their problems. Therefore, one may wonder whether the two paths of development will eventually converge into one system by merging both types and modules as first-class values into the programming language (cf. section 4.3). The major problem on this way is of course the type system, in particular, the isolation of statically computable information from dynamic computations, but some of the problems have been solved already. And if we were to include types into our language, our chosen design principles would leave us no choice: any feature important enough to be included into the design at all has to be fully integrated with all parts of the language. We would have to give full civil rights to types, including abstraction over types, modules of types, and interactions for types.

8.3 Persistent systems

All of these directions are interesting, and some of them have already seen some work on which further research could be based. However, the most promising direction for further work in our opinion stems from the extension of input/output to all kinds of expressions. This directly connects functional programming to the large body of research results and practical experience collected in the area of persistence (cf. [AM95] for a recent survey). *Persistence* is the lifetime of data objects, ranging from very short (temporary objects) to very long (objects in a database). *Orthogonal Persistence* names a language design principle, in line with the principle of data type completeness, by which the possible range of lifetimes should be the same for all kinds of objects. In particular, the treatment of data objects should be independent of their actual lifetime. Originally intended to remove the separation between programming languages (with poor support for objects of long lifetime) and database languages (with poor general programming capabilities), research on persistent languages and systems has led to a completely different view of programming environments. Not only have programming languages been developed that operate uniformly over data of different lifetime, but objects of all types – including procedures – can be stored in *persistent stores*. Over time, the separation between programs and the environment in which they are stored and executed is removed, leading to fully integrated

programming environments.

Apart from historical reasons, persistent languages (some of which are discussed in [Cla91]) have an obvious need to control the state of the persistent store and thus for a state-based programming paradigm. Typically, each persistent store has an entry point, the *root of persistence*, and all data objects reachable from this root are persistent. Programs connect to a store and may then access and update the structures reachable from the root of this store as if these were part of the programs local memories. If a sequence of updates is explicitly *committed*, or if a program closes its connection to a store, the persistent store is brought up to date. The movement of (parts of) data objects is usually treated implicitly in the runtime system, freeing programmers from tedious and error-prone work, and great attention has been paid to the development of efficient persistent stores.

Most of the goals of persistence research, e.g., better integration of programming languages and databases, the development of fully integrated data environments (theme of two successive ESPRIT Basic Research Actions [FID92]), or the use of persistent store technology as a foundation for integrated software development environments (cf. [KM97]) are of immediate practical relevance. It is therefore imperative to connect functional programming languages to the results of this very active research community, and some initial work has been done in the ML community [Har86, Mat88, NW92]. However, as long as the support for state-based computations in purely functional languages was underdeveloped, it was simply not practical to combine these languages with persistent stores. Consequently, only very little work has been done in this promising direction so far.

The first notable exception is Staple [MD91, McN93], a lazy functional language and its programming environment (STAPLE is an acronym for Statically Typed Applicative Persistent Language Environment). In Staple, modules may be added to a persistent store using commands on the operating system level and programs may access this store using the request/response-stream model. A dynamic type `Any` with suitable projection and injection operations is used to reconcile dynamic load and store of expressions with an otherwise static type system. The disadvantages of stream-based input/output-models have been discussed in chapter 3, and the two uses of the store are completely separated as modules are not values of the programming language in Staple. Furthermore, the projection of objects from type `Any` to a statically known type may fail with a type-error at runtime (no alternative code can be provided in the program for the case that a dynamic type does not match the expected static type).

More recently, Davie and Hammond [DH96] have proposed to integrate lazy functional programming, hyperprogramming and persistence into a program development environment (the term *hyperprogramming* is used in the persistence research community to describe a form of programming in which program source text may statically link to objects in a persistent store; it has been chosen by

analogy to hypertext, i.e., text that may contain links to other texts). Influenced by the module aspect of Staple, the proposal focuses on program construction in a persistent environment, defining an interactive functional hyperprogramming system on top of a persistent store. The hyperprogramming system controls the store and allows to create and manipulate functional program fragments (called *hypersheets* by analogy with spreadsheets) in this store via a graphical user interface. The functional language itself (probably Haskell) has no explicit access to the store. The proposal thus avoids the problem of program-controlled manipulations of the store and provides only a unidirectional integration: while functional programs are expected to reside and run in a persistent store, the store cannot be explicitly accessed by functional programs. Similar to Staple modules, hypersheets are not data structures of the programming language but of the (hyper-)programming environment.

While investigating the possibilities of writing an operating system in the lazy functional language Clean [PvE97], Pil has also discovered that the restrictions of input/output-systems in current purely functional languages contrast sharply with the claim that functions are first-class citizens in these language. He focuses on the typing aspects and judges that 'the file system [in most functional languages] is poorly typed at best and some classes of objects, in particular functions, cannot be stored on disk at all'. As a prerequisite for an extended input/output-system for Clean, called *First Class File I/O*, he develops a variant of dynamic types to provide an interface between statically and dynamically typed parts of a program. The type system and his plans for implementing the storage of functions in files are described in [Pil96].

Interestingly, database researchers, in search for expressive database programming languages, have invented their own functional programming languages. In the design of PFL (Persistent Functional programming Language), even referential transparency was a major design goal. Two variants of the language have been described: the first one uses a variant of result continuations to provide safe access to a persistent store [Sma93], the second one ensures safety of destructive store updates through a linear type system [SS95]. Both function definitions (sets of equations) and relations (sets of values of homogeneous type) are stored in a database and may be updated at runtime using one of the mechanisms listed above (in [Sma93], even type definitions are stored in the database and may be modified). However, the program component of the database is an unstructured set of equations, and no constructs for modular programming are described. Only the relation part of the database can be queried, and the types of values in relations have to be first-order (no functions allowed), probably even monomorphic. Therefore, the design is based on the separation between programs and first-order data, even though both are stored in one database. Other persistent languages exist that support both orthogonal persistence and first-class programs, but do not strive for referential transparency [MS92, MBC+96]. Similar to languages in the ML family, they allow both functional and imperative programming styles.

They provide a wealth of research results and implementation experience in topics such as type systems, programming environments, persistent store technology, and language design (for detailed information, cf. [Tyc97a, Tyc97b, Nap88]) that only needs to be harvested and adapted for use in purely functional languages.

In sum, the major prerequisites needed to support orthogonal persistence in purely functional languages are available: direct manipulation of persistent stores can be replaced by using the now common input/output-facilities to communicate with these stores, and the additional problems of typed languages can be solved if static typing (no typing at runtime) is replaced by strong typing. To these ends, input/output has to be extended to all kinds of expressions, and type systems have to provide both static and dynamic typing, as well as an interface between the two. However, even if these necessary prerequisites were standard in modern functional languages (which they are not yet), we would still have a backlog of several years with respect to the advanced aspects of orthogonal persistence. This estimated backlog is alarmingly large, especially if compared with the prevailing idea that functional languages are at the forefront of programming language development. The estimate is based the fact that, as early as 1984, Atkinson and Morrison [AM84] did already describe how persistent languages can support modular programming if they provide first-class procedures, thus having all essential ideas in place, albeit in the hostile framework of a procedural language. The competitive advantages manifest themselves in various areas of persistent systems, most notably in implementation experience (for instance, the file system is usually a poor replacement for an efficiently implemented persistent store). A second major advantage is the practical experience with persistent systems over longer periods of time (which led to the interest in integrated programming environments). On the positive side, the research results in orthogonal persistence are well documented, enabling functional language designers and implementors to build on the existing experience, and general language design and implementation issues (development of advanced type systems, efficient implementation of higher-order abstractions, etc.) have been pursued concurrently in both research areas. Therefore, a backlog estimate of a few years (instead of a full decade) is reasonable[5].

It should be noted that the advantages of persistent languages are not in obscure research areas but in areas of immediate practical relevance: support for the manipulation of large databases of long-lived data and support for integrated programming environments. While high-publicity languages such as Java [Jav95] are rapidly embracing persistence technology [PJa96, For95, PJW96, PJW97], the lack of support for database programming in current functional languages is getting increasingly important as a (negative) decision factor. The problem is

[5]Even with extended input/output-facilities and type systems, some of the advanced results cannot be adopted without further basic research. This is especially true for the reflective language features that are provided in some persistent languages to address issues such as system evolution [KCMS96] and dynamic typing [MM93, section 6.3].

aggravated by a tendency towards larger and larger accumulations of long-lived data, e.g., in the world-wide web, that need to be administrated. This simple fact alone could seriously hamper the transition of functional programming languages from research vehicles to practical tools, and the situation is hardly better for programming environments. The development of adequate programming environments for (purely) functional languages has so far been regarded as not a research topic, and has thus received very little attention, even though it is considered a major factor in practice. This leads to a vicious cycle: the development of suitable programming environments is assumed to be a task for commercial language implementations, which are not provided unless there is a large base of potential customers, i.e., professional software developers who are willing to pay for a language implementation with a full-featured environment. Professional software developers, however, cannot afford to invest time and money into a language for which essential tools do not exist. Experience with persistent systems has not only shown that orthogonal persistence is a solid foundation for the construction of software development environments [KM97], but also that these environments *do* generate new research problems [AM95]. In conclusion, orthogonal persistence offers tested solutions to the problems of database programming and encouraging perspectives for the problems of programming environments. Given that these topics are crucial to the success of a programming language in practice, we see no excuse for not following the principle of orthogonal persistence in the design of modern functional languages.

Chapter 9

Summary and Conclusions

The key idea of this thesis is to present the extension of a purely functional language with facilities for input/output and modular programming as a language design process. Starting from object transformations as a natural model of computations, we briefly develop functional languages as a means for declarative programming. Compared to general transformation systems, functional languages are slightly less flexible, but do not burden the daily programming task with concerns about fundamental system properties such as confluence. If properly designed on top of a suitable calculus, these languages compensate for the restrictions they impose with a refined theory for reasoning about programs. In particular, properties such as Church-Rosser and referential transparency can be guaranteed by the language designs, so that they cannot inadvertently be invalidated by programmers (chapter 2). Having thus embedded functional languages into our design framework, we collect the available design options regarding language support for input/output (chapter 3) and modular programming (chapter 4). After these preliminaries, we develop our language design, giving a formal definition of the language (chapter 5), a discussion of its support for modular programming techniques (chapter 6), and an abstract specification of our implementation (chapter 7). Finally, we show several options for further work and connect our work to research on type systems and on orthogonally persistent languages and systems (chapter 8).

When we started to take the language design aspect of our work seriously, we were surprised to find essential support in language design principles collected in the late 1970s. Functional languages have been influenced deeply by developments in the formal semantics of programming languages, and it is only reasonable to base the (re-)design of functional languages on semantic considerations, but it is interesting that characteristics of functional languages can be condensed into a few principles, which were not even developed for this purpose. The definitions of the three principles of abstraction, correspondence, and data type completeness given in the introduction to this thesis are obviously informal and leave room for different interpretations. Moreover, the purpose of the princi-

ples is easily defeated by malevolent interpretations of their defining terms. Our benevolent interpretation has turned the principles into essential tools for our language design process, but it would certainly be worthwhile if their definitions could be formalized (putting their benefits on firm ground).

In contrast to the helpful work on general language design, we found essentially no comparative surveys of design options in three areas of research that are relevant to our particular language design problem. These areas are input/output in functional languages, language support for modular programming, and the interactions between these two areas and type systems. There exist several in-depth surveys that cover large sub-areas of type system research, but the field is currently too diverse to allow for any comprehensive surveys. Unfortunately, several issues of typing language constructs for input/output and modular programming are located at the very frontier of active research on type systems, and are therefore not covered by the existing surveys (some of the issues are summarized in section 8.2). As a consequence, we decided to put aside the issues of type systems for the present thesis, and have compiled surveys of the remaining two areas that are not exhaustive, but have sufficed as a foundation for our design decisions.

In the area of input/output, several more or less detailed historical accounts of the field exist which also accurately describe some of the problems of different input/output-schemes. However, when it comes to a comparison of the different approaches, these papers are often biased either to support the particular new approach they introduce or to retain one of the well-established approaches in spite of its shortcomings. Also, the comparisons are usually informal, rendering an evaluation difficult, and even if formal accounts are given (e.g., translations between input/output-schemes in [HS89], or the operational semantics and equivalence proofs in [Gor94]), these are not in a form that could readily be employed to address the issues of language design. Nevertheless, the existing work enables us to develop a formal presentation that reduces the various input/output-schemes to their essence: the integration of context-sensitive transformations into the formerly context-free world of functional programming. Formally, the various schemes differ mainly in the restrictions which they impose on permissible contexts for interactions, but the consequences of these differences on programs that engage in interactions are considerable (as discussed in chapter 3).

The field of language support for modular programming is simply too diverse to allow for a comprehensive survey of the various language constructs and programming styles that have been proposed so far, and the problems are aggravated by the lack of a common basis for comparisons. Since we cannot build on existing work in this case, we present the relevant ideas in their historical context in chapter 4. This historical survey provides the foundation for our design decisions, but it cannot produce the necessary confidence in the final design. Therefore, we evaluate the support for the most common modular programming techniques in our language in chapter 6, establishing pragmatic a posteriori support for our design principles. Hopefully, the discussion in section 6.4 can be taken as a start-

ing point for a classification of language support for modular programming, but further work is required to develop the ideas presented in this section into a basis for a comprehensive comparison.

The major contribution of this thesis is the design of a functional programming language with explicit support for interactions with an external environment, and with implicit support for modular programming. The language is no longer purely functional, but it is still a pure language in that the integration of context-sensitive transformation rules does not affect reasoning about context-free program transformations. The overall design goal of simplicity through generality has been achieved for the extended language, and the conformance of the functional core language with the principles of abstraction, correspondence and data type completeness has been preserved for the extended language. In other words, functions, frames, and interactions are first-class data objects, abstraction is provided over all these categories, and to each declarative form of abstraction there is a corresponding parametric form. The first-class status of data objects has been extended to include participation in interactions, which allows program building blocks (modeled, e.g., as frames containing functions or interactions) to be stored in the file system and retrieved from there to become parts of other programs. Both storage and retrieval of expressions are simply interactions of programs with the file system, lifting the level of tools for program construction and maintenance to the level of programs written in the extended language.

While solving our original problems, this language design also opens some new issues regarding the interactions between the external long-term store and the programming language. For instance, should the file system be described as a data structure of the language or as some external structure (which is what we have done in chapter 5)? In other words, the questions are how tight the integration of the long-term store and the programming language should be, and how fully integrated programming systems (language + store) could look like. Fortunately, this is exactly the kind of questions that have been investigated in the research area of persistent systems (cf. section 8.3). Therefore, our language design connects the research areas of functional languages and persistent systems, and it should be possible to translate the results of research on persistent systems with imperative languages to persistent systems with functional languages.

The connection to research on persistent systems is the most promising area for further work, but the idea that functional languages can be characterized on the basis of semantic design principles, to which these languages conform more thoroughly than conventional languages, merits further investigations, too. The surveys of language support for input/output and modular programming in functional languages are also contributions of this thesis (though neither of the surveys covers its area exhaustively), and the preliminary results of section 6.4, in particular the emphasis on abstraction instead of specific language constructs, suggest that the currently very diverse lines of research and development in the field of language support for modular programming do have a common basis.

Bibliography

[ABC+83a] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 4:360–365, 1983.

[ABC+83b] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. PS-algol: A Language for Persistent Programming. In *10th Australian National Computer Conference, Melbourne, Australia*, pages 70–79, 1983.

[Ach96] Peter Achten. *Interactive Functional Programs - Models, Methods, and Implementation.* PhD thesis, Nijmegen, February 1996.

[ACPR94] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic Typing in Polymorphic Languages. Technical Report DEC-SRC-120, Digital Equipment Corporation, Systems Research Centre, Jan 1994.

[AE68] Jr. Arthur Evans. PAL – a language designed for teaching programming linguistics. In *Proceedings of the 23rd ACM National Conference*, pages 395–403, Princeton, New Jersey, 1968. Brandin Systems Press.

[AM84] M. P. Atkinson and R. Morrison. Persistent First Class Procedures are Enough. In M. Joseph and R. Shyamasundar, editors, *Proceedings of the 4th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 181 of *LNCS*, pages 223–240. Springer, 1984.

[AM95] M.P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. Technical Report FIDE/95/121, ESPRIT Basic Research Action, Project Number 6309—$FIDE_2$, 1995. Appeared in VLDB Journal, 4(3):319-401, 1995.

[AP95] Peter Achten and Rinus Plasmeijer. The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, January 1995.

[Aug93] Lennart Augustsson. Implementing Haskell overloading. In *Functional Programming and Computer Architecture*, 1993.

[Bac78] J. Backus. Can Programming be Liberated from the von Neumann Style: A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[Bar96a] Henk Barendregt. Applications of the Lambda Calculus. Talk given at the yearly colloquium of the Department of computer science in Kiel, June 1996.

[Bar96b] Henk Barendregt. The impact of the lambda calculus in logic and computer science. `ftp://ftp.cs.kun.nl/pub/CompMath.Found/church.ps.Z`, October 1996.

[Ber76] K.J. Berkling. A Symmetric Complement to the Lambda Calculus. Technical report, Gesellschaft für Mathematik und Datenverarbeitung mbH, Bonn, September 1976. ISF-76-7.

[BF82] Klaus Berkling and E. Fehr. A Consistent Extension of the Lambda Calculus as a Base for Functional Programming Languages. *Information and Control*, 55(1-3):89–101, 1982.

[BL84] R. Burstall and B. Lampson. A Kernel Language for Modules and Abstract Data Types. Technical Report 1, September 1984. Also in 'Semantics of Data Types', Springer LNCS 173; revised version appeared in Information and Computation, Vol. 76, 1988.

[BS95] Erik Barendsen and Sjaak Smetsers. Uniqueness Type Inference. In M. Hermenegildo and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs*, volume 982 of *LNCS*. Springer Verlag, 1995.

[Car84] Luca Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, pages 51–67. Springer-Verlag, 1984. LNCS 173.

[CBC+90] R.C.H. Connor, A.B. Brown, Q.I. Cutts, A. Dearle, R. Morrison, and J. Rosenberg. Type Equivalence Checking in Persistent Object Systems. In A. Dearle, G.M. Shaw, and S.B. Zdonik, editors, *Implementing Persistent Object Bases*, pages 151–164. Morgan Kaufmann, 1990.

[CD93] Charles Consel and Olivier Danvy. Partial Evaluation: Principles and Perspectives. `ftp://ftp.cis.ksu.edu/pub/CIS/Danvy/tutorial-PE.ps.Z`, 1993.

[CF74] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland Publishing Company, Amsterdam, third printing edition, 1974.

[Chu51] A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951. (second printing, first appeared 1941).

[Cla91] Stewart M. Clamen. Persistent Programming Languages - A Survey. Technical Report CMU-CS-91-155, School of Computer Science, Carnegie Mellon University, 1991.

[Cli91] Revised(4) Report on the Algorithmic Language Scheme. Technical report, 1991.

[CM94a] Luca Cardelli and John C. Mitchell. Operations on Records. In *[GM94]*, chapter 9, pages 295–350. 1994.

[CM94b] P. Cregut and D. MacQueen. An implementation of higher-order functors. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 1994.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[Dam96] Laurent Dami. Functional Programming with Dynamic Binding. Technical report, Centre Universitaire d'Informatique, University of Geneva, August 1996. (in: Object Applications, D. Tsichritzis (Ed.)).

[Dam97] Laurent Dami. A Lambda-Calculus with Dynamic Binding. *to appear in Theoretical Computer Science 192(2), special issue on Coordination, Feb 1998*, 1997.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[DD85] James Donahue and Alan Demers. Data Types Are Values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[DDH72] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*, volume 8 of *A.P.I.C. Studies in Data Processing*. Academic Press, 1972.

[Der93] Nachum Dershowitz. A Taste of Rewrite Systems, 1993. Repaired version of *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Lecture Notes in Computer Science 693, 199-228 (1993).

[DG87] L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170, Berlin, Heidelberg, New York, Tokyo, 1987. Springer-Verlag.

[DH96] Tony Davie and Kevin Hammond. Functional Hypersheets. In Werner Kluge, editor, *1996 Workshop on the Implementation of Functional Languages*, pages 39–48, 1996.

[Dij72] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[DK76] Frank DeRemer and Hans H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, SE-2(2):80–76, June 1976.

[DMN70] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. SIMULA - Common Base Language. Technical Report S-22, Norwegian computing center, Oslo, October 1970. revised edition of S-2.

[DN66] Ole-Johan Dahl and Kristen Nygaard. SIMULA – an ALGOL-Based Simulation Language. *Communications of the ACM*, 9(9):671–678, September 1966.

[eBB+97] Peter Naur (ed.), J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vaquois, J.H. Wegstein, A. van Wijngaarded, and M. Woodger. Revised Report on the Algorithmic Language Algol 60. In Peter O'Hearn and Robert D. Tennent, editors, *Algol-like Languages*, chapter 1, pages 19–49. Birkhäuser, 1997. First appeared in Communications of the

ACM 6(1):1-17, The Computer Journal 5:349-67, and Numerische Mathematik 4:420-53, 1963.

[FID92] ESPRIT Basic Research Action FIDE II. WWW home page: `http://www.dcs.gla.ac.uk/fide/default.html`, 1992. (FIDE: Fully Integrated Data Environment, PAS: Persistent Application Systems).

[For95] The Forest Project. `http://www.sunlabs.com/research/forest/`, 1995. Sun Microsystems Laboratories.

[FW76] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In Michaelson and Milner, editors, *Automata, Languages and Programming*. Edinburgh University Press, 1976.

[Gär91] D. Gärtner. *π-RED$^+$: ein interaktives codeausführendes Reduktionssystem zur vollständigen Realisierung eines angewandten λ–Kalküls.* PhD thesis, Inst. für Informatik und praktische Mathematik, Universität Kiel, 1991.

[GJ96] Benedict R. Gaster and Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996.

[GK96] D. Gärtner and W.E. Kluge. π-RED$^+$: An interactive compiling graph reduction system for an applied λ-calculus. *Journal of Functional Programming*, 6(5), September 1996.

[GLSS76] Jr. Guy Lewis Steele and Gerald Jay Sussman. Lambda: The Ultimate Imperative. AI Memo 353, MIT AI laboratory, March 1976.

[GM94] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. Foundations of Computing. MIT Press, 1994.

[GMS77] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. Early Experience with Mesa. *Communications of the ACM*, 20(8):540–553, August 1977.

[Gor94] Andrew D. Gordon. *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Har86] Robert E. Harper. Modules and Persistence in Standard ML. Technical report, University of Edinburgh, Dept. of Computer Science, September 1986. (ECS-LFCS-86-11).

[HC94] Jonathan M. D. Hill and Keith Clarke. An introduction to category theory, category theory monads, and their relationship to functional programming. Technical Report QMW-DCS-681, Department of Computer Science, Queen Mary and Westfield College, August 1994.

[Hen80] Peter Henderson. *Functional Programming – Application and Implementation*. Series in Computer Science. Prentice/Hall International, 1980.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[HHJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[HJHM76] Peter Henderson and Jr. James H. Morris. A Lazy Evaluator. In *3rd ACM Symposium on Principles of Programming Languages*, pages 95–103, 1976.

[HL94] Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. Technical Report CMU-CS-FOX-93-04, School of Computer Science, Carnegie Mellon University, January 1994. published in Proc. 21st ACM Symposium on Principles of Programming Languages.

[HMM86] Robert E. Harper, David MacQueen, and Robin Milner. Standard ML. Technical report, University of Edinburgh, Dept. of Computer Science, March 1986. (ECS-LFCS-86-2).

[HMST92] Kevin Hammond, Dave McNally, Patrick M. Sansom, and Phil Trinder. Improving Persistent Data Manipulation for Functional Languages. In J. Launchbury and P.M. Sansom, editors, *Glasgow FP Workshop*, pages 72–84. Springer Verlag, 1992.

[Hoa68] C. A. R. Hoare. Record Handling. In F. Genuys, editor, *Programming Languages, NATO Advanced Study Institute*, pages 291–347. Academic Press, London, 1968. (proceedings of a Summer School held in Villard-de-Lans, 1966).

[HPF97] Paul Hudak, John Peterson, and Joseph H. Fasel. A Gentle Introduction to Haskell. Tutorial, March 1997.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus*, volume 1 of *London Mathematical Society – Student Texts*. Cambridge University Press, 1986.

[HS89] Paul Hudak and Raman S. Sundaresh. On The Expressiveness of Purely Functional I/O Systems. Technical report, Dept. of Computer Science, Yale University, 1989.

[Hud89] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Hue90] Gérard Huet, editor. *Logical Foundations of Functional Programming*. University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

[Hug82] R. J. M. Hughes. Super combinators - a new implementation technique for applicative languages. In *ACM Symposium on Lisp and functional programming*, August 1982. see also Oxford PRG-28.

[Hug90] John Hughes. Why functional programming matters. In *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[Ing78] Daniel H. H. Ingalls. The Smalltalk-76 Programming System Design and Implementation. In *5th ACM Symposium on Principles of Programming Languages*, pages 9–15, 1978.

[Jag94] Suresh Jagannathan. Dynamic Modules in Higher-Order Languages. In *International Conference on Computer Languages*. IEEE, 1994.

[Jav95] Java Home Page. `http://java.sun.com/`, 1995.

[JJ96] Simon Peyton Jones and Mark Jones. First-class modules for component-based programming. `http://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/first-class-modules.ps.gz`, April 1996. project proposal - project will start in mid 1997.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type Classes: an exploration of the design space. In *ACM SIGPLAN Haskell Workshop*, June 1997.

[Jon95] Mark P. Jones. From Hindley-Milner Types to First-Class Structures. In *Proceedings of the Haskell Workshop*, La Jolla, California, June 1995. Yale University Research Report YALEU/DCS/RR-1075.

[Jon96] Mark P. Jones. Using Parameterized Signatures to Express Modular Structure. In *Proceedings of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 21-24 1996.

[Jon97] Mark P. Jones. First-class Polymorphism with Type Inference. In *Proceedings of the Twenty Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 15-17 1997.

[JS89] S.B. Jones and A.F. Sinclair. Functional Programs and Operating Systems. *The Computer Journal*, 32(2):162–174, 1989.

[Kah93] Stefan Kahrs. First-class polymorphism for ML. In Donald Sannella, editor, *Programming Languages and Systems – European Symposium on Programming 1994*, volume 788 of *Lecture Notes in Computer Science*, pages 333–347. Springer-Verlag, April 1993.

[KCMS96] G.N.C. Kirby, R.C.H. Connor, R. Morrison, and D. Stemple. Using Reflection to Support Type-Safe Evolution in Persistent Systems. Technical Report CS/96/10, University of St. Andrews, 1996.

[KdRB91] Gregor Kiczales, Jim des Rivières, , and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kic92] Gregor Kiczales. Towards a New Model of Abstraction in the Engineering of Software. In *IMSA'92 Workshop on Reflection and Metalevel Architectures*, 1992.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, Xerox Palo Alto Research Center, February 1997.

[Klo90] Jan Willem Klop. Term Rewriting Systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1990.

[Klu92] W.E. Kluge. *The Organization of Reduction, Data Flow and Control Flow Systems*. MIT Press, 1992. ISBN 0-262-61081-7.

[Klu94] Werner E. Kluge. A User's Guide for the Reduction System π-RED. Technical Report 9419, Inst. für Informatik und praktische Mathematik, Universität Kiel, December 1994.

[KM97] G.N.C. Kirby and R. Morrison. Orthogonal Persistence as an Implementation Platform for Software Development Environments. Technical Report CS/97/6, University of St. Andrews, 1997.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988.

[KR94] Samuel N. Kamin and Uday S. Reddy. Two Semantic Models of Object-Oriented Languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, chapter 13, pages 463–495. MIT Press, 1994.

[KS86] W. Kluge and C. Schmittgen. Reduction languages and reduction systems. In M. Vanneschi and P. Treleaven, editors, *Proceedings of the Advanced Course on Future Parallel Computers*, volume 272 of *LNCS*, pages 153–184, Pisa, Italy, June 1986. Springer.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[Lan63] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[Lan65] P. J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Parts I/II. *Communications of the ACM*, 8(2/3):89–101:158–165, February/March 1965.

[Lan66] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[Lea93] Gary T. Leavens. Introduction to the Literature On Programming Language Design. Technical Report TR 93-01b, Department of Computer Science, Iowa State University, January 1993. revised January 1994 and February 1996.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st symp. Principles of Programming Languages*, pages 109–122. ACM press, 1994.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd symp. Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997. CMU-CS-97-122.

[Lis92] Barbara Liskov. A History of CLU. Technical report, Laboratory for Computer Science, MIT, April 1992.

[Low78] James R. Low. Automatic Data Structure Selection: An Example and Overview. *Communications of the ACM*, 21(5):376–385, May 1978.

[LZ74] Barbara Liskov and Stephen Zilles. Programming With Abstract Data Types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[Mac85] D. MacQueen. Modules for Standard ML. In *Standard ML [HMM86]*. 1985.

[Mac86] David MacQueen. Using Dependent Types to Express Modular Structure. In *13th ACM Symposium on Principles of Programming Languages*, pages 277–286, 1986.

[Mac92] David B. MacQueen. Reflections on Standard ML. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *LNCS*. Springer Verlag, 1992.

[MAE+66] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Mass., 1966. second edition.

[Mat85] David C. J. Matthews. Poly and Standard ML. *SIGPLAN Notices*, 20(9):52–76, September 1985.

[Mat88] D. C. J. Matthews. An Overview of the Poly Programming Language. In *Data Types and Persistence*. Springer Verlag, 1988.

[MBC+96] R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby, and D.S. Munro. Napier88 Reference Manual (Release 2.2.1). Technical report, University of St Andrews, 1996.

[McC60] John McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine (Part I). *Communications of the ACM*, 3(4):184–195, 1960.

[McC81] John McCarthy. History of Lisp. In Richard Wexelblat, editor, *History of Programming Languages*. Academic Press, 1981.

[MCC+93] R. Morrison, R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby, and D. Stemple. Mechanisms for Controlling Evolution in Persistent Object Systems. *Journal of Microprocessors and Microprogramming*, 17(3):173–181, 1993.

[McN93] D McNally. *Models for Persistence in Lazy Functional Programming Systems*. PhD thesis, Dept. of Computer Science, University of St Andrews, 1993. Technical Report CS/93/9.

[MD91] David J. McNally and Antony J. T. Davie. Two Models for Integrating Persistence and Lazy Functional Languages. *ACM SIGPLAN Notices*, 26(5), May 1991.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit90] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 365–458. North-Holland, New York, N.Y., 1990.

[MJ95] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science 925, pages 228–266. Springer-Verlag, 1995.

[MM93] B. Mathiske F. Matthes and S. Müßig. The Tycoon System and Library Manual. DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Germany, December 1993. (Revised 19-jan-1996).

[Mog89a] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh Univ., Dept. of Comp. Sci., June 1989. Lecture Notes for course CS 359, Stanford Univ.

[Mog89b] E Moggi. Computational lambda-calculus and monads. In *Proceedings of the Logic in Computer Science Conference*, 1989.

[Mor73] James H. Morris Jr. Protection in Programming Languages. *Communications of the ACM*, 16(1):15–21, January 1973.

[Mor79] Ronald Morrison. *ON THE DEVELOPMENT OF ALGOL*. PhD thesis, Department of Computational Science, University of St Andrews, December 1979.

[MP85] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. In *ACM Symposium on Principles of Programming Languages*, 1985.

[MS92] F. Matthes and J.W. Schmidt. Definition of the Tycoon Language - A Preliminary Report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, October 1992. (Revised 17-aug-1995).

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT, 1991. ISBN 0-262-63137-7.

[MT94] David B. MacQueen and Mads Tofte. A Semantics for Higher-order Functors. In *5th European Symposium on Programming, LNCS 788*, April 1994. pp. 409-423.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT, 1990. ISBN 0-262-63132-6.

[Nap88] Napier88. `http://www-fide.dcs.st-and.ac.uk/Info/Napier88.html`, 1988.

[NW92] Scott M. Nettles and Jeannette M. Wing. Persistence + Undoability = Transactions. In *Hawaii International Conference on Systems Science 25*, January 1992. (also CMU-CS-91-173, August 1991).

[OL96] Martin Odersky and Konstantin Läufer. Putting Type Annotations to Work. In *23rd ACM Symposium on Principles of Programming Languages*, January 1996.

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Per91] Nigel Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, London, 1991.

[PGF96] Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjörn Finne. Concurrent Haskell. In *Conference Record of the Twenty Third ACM Symposium on Principles of Programming Languages*, pages 295–308. ACM, 1996.

[PH96] John Peterson and Kevin Hammond. Report on the Programming Language Haskell A Non-strict, Purely Functional Language (Version 1.3). Technical report, Haskell comittee, May 1996.

[Pil96] Marco Pil. First Class File I/O. In Werner Kluge, editor, *1996 Workshop on the Implementation of Functional Languages*, pages 341–349, 1996.

[PJa96] The PJava Project, Orthogonal Persistence for Java. `http://www.dcs.gla.ac.uk/pjava/`, 1996.

[PJW93] S. L. Peyton Jones and P. Wadler. Imperative Functional Programming. In *Proc. 20th Symp. on Principles of Programming Languages*. ACM, January 1993.

[PJW96] First International Workshop on Persistence and Java(tm) (PJW1). `http://www.sunlabs.com/research/forest/UK.Ac.Gla.Dcs.PJW1.pj1.html`, September 1996. Drymen, Scotland.

[PJW97] Second International Workshop on Persistence and Java(tm) (PJW2). `http://www.sunlabs.com/research/forest/COM.Sun.Labs.Forest.PJava.PJW2.pjw2.html`, August 1997. Half Moon Bay, San Francisco Bay Area, California.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PvE97] Rinus Plasmeijer and Marko van Eekelen. The Concurrent Clean Language Report – version 1.2 (draft). Technical report, HILT – High Level Software Tools B.V. and University of Nijmegen, March 1997.

[Rat97] Carsten Rathsack. . PhD thesis, Inst. für Informatik und praktische Mathematik, Universität Kiel, 1997. forthcoming.

[Rey70] John C. Reynolds. GEDANKEN – A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. *Communications of the ACM*, 13(5):308–319, May 1970.

[Rey93] John C. Reynolds. The Discoveries of Continuations. *Lisp and Symbolic Computation*, 6(3/4):233–248, 1993.

[RT83] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 26(1):84–89, January 1983.

[Sch71] Stephen A. Schuman, editor. *International Symposium on Extensible Languages*. ACM SIGPLAN Notices, 6(12), December 1971.

[Sch93] Wolfgang Schreiner. Parallel Functional Programming, An Annotated Bibliography (2nd Edition). Technical report, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, 1993.

[Sha81] Mary Shaw, editor. *ALPHARD: Form and Content*. Springer-Verlag, 1981.

[Sma93] C. Small. A functional approach to database updates. *Information Systems*, 18(8), 1993.

[SS72a] J. E. Stoy and C. Strachey. OS6 — an experimental operating system for a small computer. I. general principles and structure. *The Computer Journal*, 15(2):117–124, May 1972.

[SS72b] J. E. Stoy and C. Strachey. OS6 — an experimental operating system for a small computer. II. input/output and filing system. *The Computer Journal*, 15(3):195–203, August 1972.

[SS90] Harald Søndergaard and Peter Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27:505–517, 1990.

[SS95] D.R. Sutton and C. Small. Extending functional database languages to update completeness. In *13th British National Conference on Databases*, 1995.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[Str67] Christopher Strachey. Fundamental Concepts in Programming Languages. unpublished lecture notes of a course given at the International Summer School in Computer Programming at Copenhagen, August 1967.

[SW74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. Technical Report PRG-11, Programming Research Group, Oxford University, January 1974.

[SW80] Mary Shaw and Wm. A. Wulf. Toward Relaxing Assumptions in Languages and Their Implementations. In Mary Shaw, editor, *ALPHARD: Form and Content*, chapter 10, pages 295–313. Springer-Verlag, January 1980.

[Taf93] S. Tucker Taft. Ada 9X: From Abstraction-Oriented to Object-Oriented. In Andreas Paepcke, editor, *ACM OOPSLA'93*, pages 127–135, October 1993.

[Tan76] Andrew S. Tanenbaum. A Tutorial on Algol 68. *Computing Surveys*, 8(2):155–190, June 1976.

[Ten77] R. D. Tennent. Language Design Methods Based on Semantic Principles. *Acta Informatica*, 8:97–112, 1977.

[Tho90] Simon Thompson. Interactive Functional Programs - A Method and a Formal Semantics. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 10, pages 249–285. Addison-Wesley, 1990.

[Tim96] Stephan Timm. Ein/Ausgabe-Operationen in KiR. Master's thesis, Inst. für Informatik und praktische Mathematik, Universität Kiel, July 1996.

[Tyc97a] Tycoon. `http://www.sts.tu-harburg.de/projects/Tycoon/entry.html`, 1997.

[Tyc97b] Tycoon-2. `http://www.sts.tu-harburg.de/projects/Tycoon2/entry.html`, 1997.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA '87, Orlando, Florida*, pages 227–241, 1987. Published as SIGPLAN Notices 22(12), December, 1987. Also published in Lisp and Symbolic Computation 4(3), Kluwer Academic Publishers, June, 1991.

[vW63] Adriaan van Wijngaarden. Generalized ALGOL. In Richard Goodman, editor, *Annual Review in Automatic Programming, Vol. 3*, pages 17–26. Pergamon Press, Oxford, 1963.

[Wad71] Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Programming Research Group, University of Oxford, September 1971.

[Wad87] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14th ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987.

[Wad92a] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2(4), 1992. (Special issue of selected papers from 6'th Conference on Lisp and Functional Programming.).

[Wad92b] P. Wadler. The essence of functional programming. In *POPL '92, Albuquerque*, 1992.

[Was80] Anthony I. Wasserman. *TUTORIAL Programming Language Design*. IEEE Computer Society Press, Los Alamitos, Calif., 1980. Initally

presented at Compsac80, The IEEE Computer Society's Fourth International Computer Software & Applications Conference, October 27-31, 1980. IEEE catalog no.: EHO 164-4.

[Weg76] Peter Wegner. Programming Languages – The First 25 Years. *IEEE Transactions on Computers*, pages 1207–1225, December 1976. reprinted in Wasserman80.

[Wel96] Joe Wells. Typability and Type Checking in the Second-Order λ-Calculus Are Equivalent and Undecidable. *submitted to APAP ??*, June 1996. (an earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science 1994).

[Wex81] Richard L. Wexelblat, editor. *History of Programming Languages.* Academic Press, 1981. Proceedings of the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978, Los Angeles, California.

[Wex93] Richard L. Wexelblat, editor. *Proceedings of the Conference on History of Programming Languages*, volume 28(3) of *ACM Sigplan Notices*, New York, NY, USA, April 1993. ACM Press.

[WH66] Niklaus Wirth and C. A. R. Hoare. A Contribution to the Development of ALGOL. *Communications of the ACM*, 9(6):413–432, June 1966.

[Wir77] Niklaus Wirth. Design and Implementation of Modula. *Software – Practice and Experience*, 7:67–84, 1977.

[Wir79] Niklaus Wirth. The Module: A System Structuring Facility in High-Level Programming Languages. In Jeffrey M. Tobias, editor, *Proceedings of the Symposium on Language Design and Programming Methodology held in Sidney, Australia*, pages 1–24. Springer Verlag, 1979.

[WWG57] Maurice V. Wilkes, David J. Wheeler, and Stanley Gill. *The Preparation of Programs for an Electronic Digital Computer.* Addison-Wesley, second edition, 1957.

[X3J94] X3J13. ANSI Common Lisp standard. Technical report, 1994.

[Xer96] Open Implementation Group, Xerox PARC. `http://www.parc.xerox.com/spl/projects/oi/`, 1996.

[Zil73] Stephen N. Zilles. Procedural Encapsulation: A Linguistic Protection Technique. *SIGPLAN Notices*, 8(9):142–146, September 1973.