UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ANTONIO SOARES DE AZEVEDO TERCEIRO

Semantics for an Algebraic Specification Language

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Daltro José Nunes
Advisor

Porto Alegre, May 2006

CIP – CATALOGING-IN-PUBLICATION

Azevedo Terceiro, Antonio Soares de

Semantics for an Algebraic Specification Language / Antonio Soares de Azevedo Terceiro. – Porto Alegre: PPGC da UFRGS, 2006.

87 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2006. Advisor: Daltro José Nunes.

1. Prosoft Environment. 2. Algebraic Specification. 3. Denotational Semantics. 4. Operational Semantics. 5. Semantic Prototyping. 6. Haskell Programming Language. I. Nunes, Daltro José. II. Título.

ACKNOWLEDGEMENTS

To Professor Daltro Nunes, who was a great advisor and contributed with all his experience and practice to my advance in the technical, professional and personal fields.

To my colleagues at the Prosoft research group for the friendship and academic cooperation.

To the Brazilian people, for funding my work through The National Council of Scientific and Technological Development (CNPq).

To Tânia, Günther, Ricardo, Rodrigo, Wally and Mrs. Fanny, for being my family in Porto Alegre.

To Machado and Lucas, for being my reference of Bahia and for being so great friends.

To my family and friends in Salvador, who missed me and who I missed so much.

To the Free Software community, for providing all that great software, without which this work wouldn't be possible, and for showing that the knowledge is much more useful for all society when it's shared.

To the people from Associação Software Livre.Org, for all the things I learned with them.

To all the trolls who feel as part of the Free Software community, for always remembering me of the "do more, talk less" principle.

To Josy, for the greatest love one can experiment and for supporting me during the realization of this work.

CONTENTS

ABSTRACT 6

RESUMO 7

1 INTRODUCTION 8
1.1 Motivation 8
1.2 Goals 9

2 FOUNDATIONS 10
2.1 Algebraic Specification Methods 10
2.1.1 Algebraic Specifications 11
2.2 The Prosoft Environment 12
2.2.1 Albebraic Prosoft 12
2.2.2 Prosoft Java 13
2.3 Formal Semantics 13
2.3.1 Denotational Semantics 13
2.3.2 Axiomatic Semantics 14
2.3.3 Operational Semantics 14
2.3.4 Action semantics 15
2.4 Final remarks 15

3 ALGEBRAIC PROSOFT BASICS 16
3.1 An overview of Algebraic Prosoft 16
3.1.1 An example 16
3.1.2 Foundations 17
3.2 Graphical representation for Composite data types 19
3.2.1 Built-in data types representation and instantiation 19
3.2.2 Instantiating built-in types in user-defined data types 21
3.2.3 Semantics' informal notions 21
3.3 Term Reduction and the ICS 22
3.4 Final remarks 23

4 PROSOFT FORMAL SEMANTICS 24
4.1 Syntax 24
4.2 Relating graphical representation of ATO's and syntax 25
4.2.1 Sets 25
4.2.2 Maps 26
4.2.3 Lists 26

4.2.4 Records . . . 26
4.2.5 Unions . . . 26
4.2.6 Multi-level trees and their instantiation . . . 26
4.3 Notation . . . 27
4.4 Semantic domains . . . 28
4.5 Semantic functions . . . 28
4.6 Instantiation . . . 31
4.7 Matching . . . 34
4.8 Auxiliary Functions . . . 36
4.9 Final remarks . . . 38

5 SEMANTICS-BASED LANGUAGE PROTOTYPING WITH HASKELL . 39
5.1 Introduction . . . 39
5.2 The Haskell programming language and semantic prototyping . . . 40
5.3 Developing semantics-based prototype implementations . . . 40
5.3.1 Abstract syntax . . . 41
5.3.2 Basic semantic domains . . . 41
5.3.3 A denotational semantics for *toy* . . . 43
5.3.4 An operational semantics for *toy* . . . 47
5.4 More elaborated prototypes . . . 50
5.5 Related work . . . 50
5.6 Final remarks . . . 51

6 THE DEVELOPMENT OF A SEMANTICS-BASED PROTOTYPE FOR ALGEBRAIC PROSOFT . . . 53
6.1 Literate Programming . . . 53
6.2 Literate Programming and the Haskell Programming Language . . . 54
6.3 The prototype . . . 55
6.4 A Semantic Prototype Framework . . . 57
6.5 Final remarks . . . 58

7 CONCLUSIONS . . . 59
7.1 Related work . . . 59
7.2 Contributions . . . 59
7.3 Limitations . . . 60
7.4 Future work . . . 60

REFERENCES . . . 62

APPENDIX A PROSOFT-REDUCE SIMPLIFIED REFERENCE MANUAL . 65

APPENDIX B BUILT-IN ATO'S . . . 69

APPENDIX C THE VIDEOCLUB EXAMPLE, REVISITED . . . 77

ABSTRACT

Prosoft is a research project at Instituto de Informática da UFRGS, developed by the research group with the same name and coordinated by Professor Daltro José Nunes. The project's goal is to develop a full software development environment, the Prosoft Environment, based on the concepts of Models, Lambda Calculus, Abstract Data Types and Object orientation.

One of the components of the Prosoft Environment is its algebraic specification language: Algebraic Prosoft. Although being the basis and theme of several works in the Prosoft research group, Algebraic Prosoft doesn't have its semantics properly defined. Works done up to now were based on operational notions and presented different interpretations of Algebraic Prosoft.

This thesis presents a denotational semantics specification for Algebraic Prosoft, comprising, among other features, its "inter-data type" communication primitive, called ICS, and its graphical notation for representing instantiations of abstract data types.

This thesis also presents a study of semantic prototyping using the Haskell programming language. The concept of Literate Programing and the proximity between lambda calculus and Haskell were crucial to the rapid development of a prototype implementation of Algebraic Prosoft, based on its specified semantics.

This thesis' main contributions include: a precise and unambiguous interpretation of Algebraic Prosoft, through a semantics specification; the definition of semantics to the ICS, a unique (to the best of our knowledge) concept that provides a message-passing mechanism between algebraic data types; a prototype implementation of Algebraic Prosoft, which can actually be used to experiment and test the Algebraic Prosoft language definition and semantics specification; results regarding semantics prototyping of both denotational and operational semantics specifications using the Haskell programming language for rapid development of semantics-based prototypes of languages.

Since a large portion of Prosoft Environment's development is done through international cooperation projects and this thesis will strongly influence its future development, the text was written in English in order to facilitate the information exchange between the Prosoft research group and its foreign partners.

Keywords: Prosoft Environment, Algebraic Specification, Denotational Semantics, Operational Semantics, Semantic Prototyping, Haskell Programming Language.

Semântica para uma Linguagem de Especificação Algébrica

RESUMO

Prosoft é um grupo de pesquisa do Instituto de Informática da UFRGS, desenvolvido pelo grupo de pesquisa homônimo e coordenado pelo Professor Daltro José Nunes. O objetivo do projeto é desenvolver um ambiente de desenvolvimento de software completo, o Ambiente Prosoft, que é baseado nos conceitos de Modelos, Cálculo Lambda, Tipos Abstratos de Dados e Orientação a Objetos.

Um dos componentes do Ambiente Prosoft é sua linguagem de especificação algébrica: o Prosoft Algébrico. Apesar de ser base e tema de diversos trabalhos no grupo de pesquisa Prosoft, o Prosoft Algébrico não tem sua semântica devidamente definida. Os trabalhos desenvolvidos até agora foram baseados em noções operacionais, e apresentam diferentes interpretações do Prosoft Algébrico.

Esta dissertação apresenta uma especificação de semântica denotacional para o Prosoft Algébrico, compreendendo, entre outras características, sua primitiva de comunicação entre tipos de dados, chamada ICS, e sua notação gráfica para representação de instanciação de tipos abstratos de dados.

Essa dissertação apresenta também um estudo sobre prototipação semântica usando a linguagem de programação Haskell. O conceito de *Literate Programming* e a proximidade entre Cálculo Lambda e Haskell foram cruciais no rápido desenvolvimento de uma implementação protótipo do Prosoft Algébrico, baseada na sua semântica especificada.

As principais contribuições dessa dissertação incluem: uma interpretação precisa e sem ambiguidades do Prosoft Algébrico, através da especificação da sua semântica; a definição de semântica para a ICS, um conceito único (até o limite do nosso conhecimento) que fornece um mecanismo de passagem de mensagens entre tipos de dados algébricos; uma implementação protótipo do Prosoft Algébrico, que pode realmente ser utilizada para experimentar e testar a definição da linguagem e a especificação da semântica do Prosoft Algébrico; resultados sobre prototipação semântica de especificações tanto de semântica denotacional quanto de semântica operacional usando a linguagem de programação Haskell para desenvolvimento rápido de protótipos de linguagens baseados na sua semântica.

Como grande parte do desenvolvimento do Ambiente Prosoft é realizado através de projetos de cooperação internacional e essa dissertação irá influenciar fortemente o seu desenvolvimento futuro, o texto foi escrito em inglês para facilitar a troca de informação entre o grupo Prosoft e seus parceiros estrangeiros.

Palavras-chave: Ambiente Prosoft, Esperificação Algébrica, Semântica Denotacional, Semântica Operacional, Prototipação Semântica, Linguagem de Programação Haskell .

1 INTRODUCTION

In this chapter, the fundamental motivations and goals for this work are presented. Section 1.1 discusses motivations for this work, and section 1.2 states the goals aimed with it.

1.1 Motivation

Prosoft is a research project at Instituto de Informática da UFRGS, developed by the research group with the same name and coordinated by Professor Daltro José Nunes.

The project's goal is to build a full Software Development Environment supporting the Software Engineer from the earliest phase of requirements gathering, to the implementation phase.

An important component of the Prosoft environment is its algebraic notation, a powerful notation for specifying Abstract Data Types, called Algebraic Prosoft.

The Algebraic Prosoft language passed through an evolutionary process. It was initially proposed by Professor Daltro Nunes in his doctoral thesis for representing program control structures in a graphical fashion, when it was called just "Prosoft". Later it was adapted to represent algebraic data types, and has being used for that purpose until the curent date. This algebraic branch of Prosoft became the base of a long-term research initiative, comprising several projects, on Data-Driven Software Engineering. Later, a primitive for inter–data type operations was added to Prosoft. More recently, this primitive for inter–data type operation was restricted to applying only monadic operations.

Although being the basis and the theme of several works in the Prosoft research group, Algebraic Prosoft doesn't have its semantics properly defined. Work done until now have been done based on operational notions, and presented different interpretations of Algebraic Prosoft. This was caused both by the natural evolution of the language and the lack of a proper formal semantics.

Formalizing Algebraic Prosoft's semantics will improve the Prosoft Environment as a whole, since as soon as one has an exact meaning for the language's constructions, Algebraic Prosoft will have adequate mathematical foundations, allowing:

- precise interpretation of its notation;
- rapid specification prototyping through a semantics-based term reduction tool;
- proof of properties over specifications.

1.2 Goals

This work proposes the construction of a denotational semantics for Algebraic Prosoft, fulfilling the existing gap in formalizing the language's constructs. This way, the specific goals of this work are:

- Specify and document a syntax for Algebraic Prosoft;
- Formalize and document an unique interpretation of Algebraic Prosoft, to serve as a "user guide" to upcoming work on the Prosoft Environment.
- Create a denotational semantics for Algebraic Prosoft comprehending its core concepts (NUNES, 2003).
- Develop a semantics-based prototyping tool for Algebraic Prosoft, featuring term reduction over user-created specifications.

Algebraic Prosoft formalization will drastically improve the Prosoft Environment, since it opens the possibility for property proving inside the Prosoft Environment, what will lead to a quality enhancement both in the development process and in the developed products.

Although proof of properties is very important, it's outside the scope of this work. Giving semantics for Algebraic Prosoft, however, is a first step towards property proving over Prosoft specifications. The author hopes that this work can act as a solid base to future work aiming property proving inside the Prosoft Environment.

2 FOUNDATIONS

This chapter presents foundation topics that were studied during the development of this thesis.

Section 2.1 introduces Algebraic Specification Methods. Section 2.2 presents the Prosoft Environment. Section 2.3 discusses Formal Semantics and give a brief overview of the most common methods. Section 2.4 summarizes this chapter.

2.1 Algebraic Specification Methods

Algebraic Specification Methods are formalisms that aim at representing the solutions for problems through algebras. Algebraic approaches privilege representing input and output of programs, rather than other concerns in software development, such as concurrency or distribution.

In Algebraic Methods, a solution to a problem consists on the definition of a many-sorted algebra: a collection of sets of data to represent program data, and operations(functions) over those sets representing programs' data transformation.

Algebraic specifications are formalisms to describe problems in terms of those algebras. They represent problems using sets of objects (terms) for problem data, and operations between those sets. Operations can either query the state of objects or create other objects based on their state.

Classical problems in Algebraic Specification (SANNELLA; TARLECKI, 1999) include:

- What is a specification?
- What does a specification mean?
- When does a program satisfy a specification?
- When does a specification guarantee a property that it does not state explicitly?
- How does one prove that?
- How are specifications structured?
- How does the structure of specifications relate to the structure of programs?
- When does one specification correctly refine another specification?
- How does one prove correctness of refinement steps?

- When do refinement steps compose?
- What is the role of information hiding?

This work focuses on answering the two first questions in the context of Algebraic Prosoft, namely:

- Define in a precise way the syntax for Algebraic Prosoft specifications.
- Give precise meaning to Algebraic Prosoft specifications.

2.1.1 Algebraic Specifications

Algebraic specifications is a broad subject. This section presents the concepts regarding Algebraic Specifications, constrained to the notions relevant to this thesis.

As said before, Algebraic Specifications describe data types in terms of algebras: possible values and operations over those values. They have two main parts: operation signatures and equations.

Signatures define the form of operations. They exhibit the operation symbol, its domain and its range:

$$op\,(\_, \_, \ldots, \_) : s_1, s_2, \ldots, s_n \rightarrow s$$

In the above signature, op is the operation symbol, s_i the domain sorts, and s the range sort. _ are placeholders for the operation arguments, and there are as many of them as sorts in the operation's domain. The form of the operation can vary from the prefixed form shown above: we can have post-fixed forms, as well as infixed ones (like in the signature $if\ \_\ then\ \_\ else\ \_ : Bool, X, X \rightarrow X$), but in this work we'll stick to the prefixed form in order to keep simplicity.

If the domain of an operation is empty, it's called a constant.

For a given sort s, the set of its terms is defined as follows:

- Every constant c with s as its range is a term of sort s.
- Every variable v, denoting some value of the sort s, is a term of sort s.
- For all operations $op\,(\_, \_, \ldots, \_) : s_1, s_2, \ldots, s_n \rightarrow s$, $op\,(t_1, t_2, \ldots, t_n)$ is a term of sort s if each t_i is a term of sort s_i.

Equations define the meaning of operations, by giving the equality of two terms, in the form $lhs = rhs$, where lhs is often a term that matches the operation signature. lhs may contain variables, and they can be referenced in rhs: as a general rule, all variables present in rhs must be present in lhs: rhs can be thought as a function of the variables present in lhs.

When an operation has no associated equations, its a generator operation: it represents a value of the sort in its range. [1]

The semantics of an algebraic specification is an algebra.

Algebraic specifications are often represented as Rewriting Systems. Given a set of equations, a Rewriting System transforms terms using the equations: whenever a term t matches the left-hand side (lhs) of an equation, it is replaced by the equation's right-hand side (rhs), substituting in rhs all variables present in lhs that matched subterms of t by those subterms.

[1] The reciprocal is not necessarily true: generator operations can have associated equations.

2.2 The Prosoft Environment

The Prosoft project's goal is to build a full Software Development Environment supporting the Software Engineer from the earliest phase of requirements gathering to the implementation phase: the Prosoft Environment.

The Prosoft Environment acts as a rich laboratory for research on Software Engineering, specifically in the fields of Formal Methods, Software Process Technology, Distance Education and Computer Supported Cooperative Work.

The Prosoft Environment comprises several components. The most important of them are Algebraic Prosoft and Prosoft Java. Several other components are built on top of Prosoft Java, extending the Prosoft Environment to specific domains (REIS, 1998a,b; SOUSA, 2003; REIS, 2002, 2003; RANGEL, 2003; FREITAS, 2005; MAIA, 2005; DAHMER, 2006).

The following subsections describe the two main components in more detail.

2.2.1 Albebraic Prosoft

The principles leading to the choice of an algebraic method as the foundation for the Prosoft Environment (NUNES, 1994) are:

- **Data-driven strategy.** Software Development is based on data structures definitions representing the data relevant to the problem. The solution is given defining operations over those data. (NUNES, 1992)
- **The concept of model.** A solution to some problem is the model of some theory.
- **Lamdba Calculus.** In Lamdba Calculus, operations with more than one argument are represented as higher-order functions. Lambda Calculus's influence can be noted in Algebraic Prosoft's semantics description.
- **Abstract Data Types.** In data structures definitions, one can use predefined composite types to build more complex and problem-specific ones.
- **Object orientation.** Reusability, provided by the concepts of messages, inheritance and polymorphism, is a characteristic that is goal of any Software Development Environment.

In Algebraic Prosoft, the main construction block is the ATO [2]. One ATO contains one specification, defining one or more sorts and operations over those sorts (i.e. a data type). An ATO can only handle objects from the sorts defined by it.

Whenever an ATO needs to operate on objects from sorts defined in other ATO's, it uses an special operation named ICS. The ICS is responsible for "calling" operations defined in other ATO's over objects that are elements of one of their sorts. We're used to define the ICS as a message bus that "connects" the different ATO's (as seen in figure 3.6).

Chapter 3 presents Algebraic Prosoft in more detail, together with an informal description of it.

[2] Portuguese acronym for Object Handling Environment ("Ambiente de Tratamento de Objetos")

2.2.2 Prosoft Java

The concept of the Prosoft Environment has been focus of research since early 90's. There were several implementations of the Prosoft Environment, and each of them presented considerable enhancements compared to the previous one.

Prosoft's first version was a single-user implementation written in Solaris-Pascal (NUNES, 1992). ATO's were implemented by writing Pascal functions, and the system as a whole needed to be compiled into a monolithic program.

After that, Distributed Prosoft was develop in a mix of Pascal and C, as described in (SCHLEBBE, 1994) and (GRANVILLE; GASPARY, 1996). This version supported distributed and cooperative work on the Prosoft Environment.

Prosoft's current implementation — Prosoft Java — was written in Java (SCHLEBBE; SCHIMPF, 1997) and features a more modern design. New ATO's can be added without the need to recompile its kernel. Distributed and cooperative work are supported through Java's standard technologies for distribution.

Prosoft Java provides a visual environment for designing Prosoft specifications, embedding ATO's implemented in Java, and creating suitable user interfaces for those ATO's. Prosoft Java is the base of several of the current research works in the Prosoft research group.

2.3 Formal Semantics

> Semantics is concerned with the meaning of phrases in a language. (WATT, 1991)

Formal semantics, or simply semantics, is concerned with assigning *precise* meaning to phrases in a language, achieving a mathematical model of the language, in terms of which one can reason about its programs. Phrases are often associated with mathematical entities, such as functions, sets, set elements, terms, tuples, etc: this way, the underlying mathematical theories of these entities can be used for reasoning about programs. Depending on the methodology applied for giving semantics, the kind of mathematical entities will vary.

Semantics is often given in terms of abstract syntax trees, which represent the structure of programs in an abstract way. In abstract syntax, we are concerned only in the structure of the program, and not in the actual syntax that would be used by a programmer to write that program. An abstract syntax is defined by a grammar, which produces the (probably infinite) set of possible programs that can be written in that language.

For each kind of node in the program's abstract syntax tree (productions in the corresponding grammar), it is given some meaning in terms of mathematical entities, as we shall see for each methodology described below. A desirable aspect of semantics is that compound constructs' semantics can be defined in terms of their components' semantics.

The following subsections describe briefly the major semantics methodologies that were studied for the development on this thesis.

2.3.1 Denotational Semantics

In Denotational Semantics, programs have their meaning expressed in terms of sets and functions. Thus, giving a denotational semantics to a language consists of associating each language construct to a number, a set, a function, etc. We say that each language construct is *denoted* by one of those mathematical entities.

An important property of denotational semantics is that the denotation of a composite construct is given in terms of the denotations of its components. This allows us, among other things, to specify precisely and concisely the semantics of languages that allow combination of several of its primitive constructs to build more complex ones.

For example, the semantics for commands can be defined in terms of a function $execute$, whose definition for loops of the type `while E do C` could be given as in figure 2.1 (WATT, 1991): the semantics of a *while* statement consists of first evaluating the condition, and if it is true, execute the body and re-execute the whole *while* statement.

```
execute ⟦while E do C⟧ =
  let execute-while env sto =
    let truth-value tr = evaluate ⟦E⟧ env sto in
    if tr
      then execute-while env (execute ⟦C⟧ env sto)
      else sto
  in
    execute-while
```

Figure 2.1: semantics definition for the **`while E do C`** construct

A great advantage of denotational semantics is that its notation can be almost transliterated into a modern functional programming language (as shown in chapter 5), what makes possible the rapid prototyping of semantics specifications.

More information on denotational semantics can be found in chapters 3, 4 and 5 of (WATT, 1991), in (SCHMIDT, 1986), and in the literature referenced on these books.

2.3.2 Axiomatic Semantics

Axiomatic semantics was initially created to proof correction of programs, and it is based on predicate logic. Only later it was shown that axiomatic semantics techniques — which at that times wasn't called this way — could be used to specify semantics of programming language constructs. (WATT, 1991)

An axiomatic semantics for language consists in a set of assertions about its properties. If we consider that in a given program written in some language, the execution of its parts changes the state of the system, then the language's axiomatic semantics is a set of invariants on the state that is preserved through the execution of the program.

More information on axiomatic semantics can be found in (GRIES, 1981) and in the literature referenced there. In (DIJKSTRA, 1976), the roots of axiomatic semantics are presented.

2.3.3 Operational Semantics

Operational semantics represents the execution of a program as a transition system, in order to model how the state of an abstract (but realistic) machine changes during the execution of programs. For that, one defines a set Γ of machine configurations, and a transition relation $\rightarrow \subseteq \Gamma \times \Gamma$, where $\gamma \rightarrow \gamma'$ means "there is a transition from configuration γ to configuration γ'".

The structure of the configurations $\gamma \in \Gamma$ vary depending on each language to which one is giving semantics. It often represents the structure of the program (which suffers transformations while the program is "run"), plus every relevant aspect of the abstract

machine that is updated during execution of programs: memory, bindings, scope, etc. Depending on the type of language — functional, imperative, object-oriented, with static bindings, with dynamic bindings, ... — to which is being given an operational semantics, the particular structure of the configurations becomes more or less complex.

For each type of language construct (expressions, commands, declarations, etc) the corresponding machine transitions are defined through natural deduction rules. Chapter 5) shows an example of operational semantics for a toy language, together with its implementation in a modern functional programming language.

More information about denotational semantics can be found in (PLOTKIN, 1981) and in the literature referenced there.

2.3.4 Action semantics

Action semantics represents a compromise between the mathematical formalism of denotational semantics and the operational notions (in terms of which most of us has the intuitive notion of a language's semantics) of operational semantics. Defined in terms of abstract data types — whose semantics is given by mathematics' algebras — and using a naming scheme for its sorts and operators that is very close to English, action semantics tries to make semantics specifications more natural.

In action semantics, the semantics of language constructs is given in terms of an abstract data type *Action*, which has several subtypes.

An advantage of using action semantics is that having the axioms of the data types used in the semantics, one can submit the term representing a program to a term rewriting system, allowing a simulation of the program's execution through the term's reduction.

More information about action semantics can be found in (WATT, 1991) and in the literature referenced there.

2.4 Final remarks

This chapter presented a short overview of the underlying theories hat serve as foundation of this work, and were object for study during the preparation of this thesis.

- Algebraic Specification methods are used to build solutions for problems in terms of algebras. They comprise operations signatures and equations.
- The Prosoft Environment is a software development environment, aiming to support the entire software development process. It uses Algebraic Prosoft as its specification notation.
- Formal Semantics is concerned with giving precise meaning to language constructs, in order to allow verifications, proofs and reasoning about programs.

Next chapter presents Algebraic Prosoft's main characteristics: the syntax of its specifications, its graphical notation for representing instantiations of data types. Near to its end, the next chapter also presents an informal notion of Algebraic Prosoft's semantics.

3 ALGEBRAIC PROSOFT BASICS

As seen is chapter 2, Algebraic Prosoft is the language used for algebraic specification in the Prosoft Environment.

This chapter presents the main principles of Algebraic Prosoft. Section 3.1 gives an overview of the language, by showing some examples of its use. Section 3.2 describes Algebraic Prosoft's graphical notation for data types, together with an informal interpretation of its semantics. Section 3.3 presents an informal notion of Algebraic Prosoft by describing how term reduction works in it. Section 3.4 summarizes the chapter.

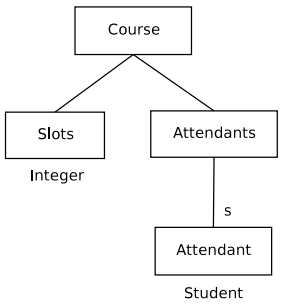
3.1 An overview of Algebraic Prosoft

To present Algebraic Prosoft's features, this section first shows a commented example of a Prosoft specification. Later in this section, we'll present the foundations for the Algebraic Prosoft notation.

3.1.1 An example

Suppose we are specifying an academic management system. A basic element of it would be *courses*: they have a maximum number of attendants, as well as a set of those attendants (which must contain no more elements than the course's maximum).

specification Courses



operations
  create_course: Integer → Course
  add_attendant: Course → Course
  attendance_list: Courses → List
  ...

In Prosoft, a specification has three parts:

The **instantiation** defines the structure of the data type. In this case, a $Course$ is a record, whose fields are the number of available slots (an integer), and some $attendants$, which form a set. Those attendants are $Students$, whose data type structure is specified in some other specification.

In the **operations** section, the specifier can extend the type (a record, in this case) by

adding new operations and their functionality, specifying the interface for the type. In our example, the operations $create\_course$, $add\_attendant$ and $attendance\_list$ were defined.

```
equations
  create_course(n) = make_course(n,emptySet)

  add_attendant(make_course(slots,attendants)) =
    if eq(length(attendant),slots)
      then error
      else make_course(slots,insert(attendants,s?))

  attendance_list(make_course(_,attendants)) =
    process_attendance_list(attendants)

  process_attendance_list(emptySet) = emptyList
  process_attendance_list(insert(set,attendant)) =
    cons(ICS(Students,get_name,attendant),
         process_attendance_list(set))
```

The **equations** section specify the semantics of each of the operations defined for the type. In our example, all the operations defined as part of the type's interface (`create_course`, `add_attendant` and `attendance_list` are specified, as well as an auxiliary one (`process_attendance_list`), that wasn't declared as part of the interface. Note the use of the ICS operation. It is used to refer to an operation in another specification.

3.1.2 Foundations

The basic building block in Algebraic Prosoft is the ATO [1]. An ATO defines one (maybe abstract) data type, by indicating the sorts that participate in the type, an interface for the type (operations and their signatures) and the semantics of operations through algebraic equations.

In Prosoft, abstract data types are specified using a graphical notation. The **instantiation** defines, based on the built-in composite types, a new sort. Figure 3.1(a), for example, shows the graphical representation of an instantiation of the *Map* abstract data type, using *Code* as its domain, *ProductInfo* as its range. The whole sort is called *Catalogue*. *Code* belongs to the sort *Integer*, while *ProductInfo* is an external sort, defined elsewhere.

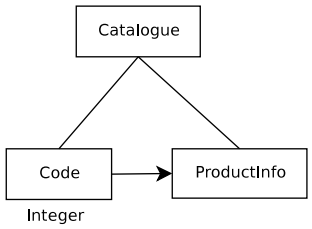
Figure 3.1(b) shows another example: Employees is an instantiation of the abstract data type *Set*, and its elements belong to the sort *Employee*.

The **operations** section defines the ATO's interface: what operations are available, an the signature of each operation. For example, the following definition specifies an operation op, with sorts $E_1, E_2, \ldots, E_n$ as its domain, and the sort E as its range.

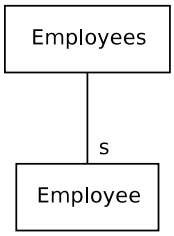
$op : E_1, E_2, \ldots, E_n \rightarrow E$

When a new specification is instantiated, the base types operations are imported, and are made available in that ATO. The specifier can extend this standard definition through the definition of new operations.

[1] Portuguese acronym for "Object Handling Environment"



(a) A mapping with *Code* as domain and *ProductInfo* as range



(b) *Employees* is a set, whose elements belong to the sort *Employee* (defined elsewhere)

Figure 3.1: Examples of instantiation using Prosoft graphical language

Each new operation is given a signature in the **operations** section, and can have its semantics specified in the **equations** section. Operations to which no semantics is given (through equations) are generators of its range sort.

The **equations** section defines the operations' semantics.

Each equation has the form $lhs = rhs$, where both lhs and rhs are terms. Each equation's lhs must match the signature of some operation in the **operations** section.

In Prosoft, operations can be monadic [2] . An operation $op : E_1, E_2, \ldots, E_n \rightarrow E$ is interpreted as $op : E_1 \rightarrow (E_2, \ldots, E_n \rightarrow E)$, as in lambda calculus. This means that op results are operations with signature $E_2, \ldots, E_n \rightarrow E$. Those operations are defined in the same ATO that defines the sort E_2.

Lets say that the sort E_2 is defined in ATO_2. The equations for op, in this example, would define, based on the state of some term t_1, from sort E_1, what operation in ATO_2 must be applied to carry on the rest of the computation, with arguments $t_2, \ldots, t_n$.

In the special case where $n = 1$ (i.e., $op : E_1 \rightarrow E$), the equations for op can be defined in terms of E's constructor operations (the simplest case), or in terms of other operations of the ATO or of other ATO's .

In the right side of the equations, it is often needed to reference operations defined in other ATO's; to do that, Prosoft provides a special operation, called ICS [3] . ICS has two forms:

- $ICS(ATO, op)$
- $ICS(ATO, op, \langle t_1, t_2, \ldots, t_n \rangle)$

The first form is reserved for n-ary operations (with $n > 1$), and represents a reference to an operation in another ATO. For example, the following equation defines $op : E_1, E_2, \ldots, E_n \rightarrow E$ in terms of $op' : E_2, \ldots, E_n \rightarrow E$:

$op(t(v_1, v_2, \ldots, v_m)) = ICS(ATO_2, op')$

The second form is used for explicit call to operations in other ATO's. For example, the following equation defines $op : E_1 \rightarrow E_2$ in terms of $op' : E' \rightarrow E_2$ (where E' is the sort of the subterm v_1)

$op(t(v_1, v_2, \ldots, v_m)) = ICS(ATO_2, op', v_1)$

Section 3.3 gives a informal notion on ICS semantics.

[2] They are not required to be monadic, but as it will be seen later, being monadic is a requisite for being accessible through ICS.

[3] Portuguese acronym for "System Communication Interface".

3.2 Graphical representation for Composite data types

Algebraic Prosoft uses a powerful graphical notation for representing composite abstract data types. Through this language, one can instantiate the available built-in composite abstract data types, as well as reuse other user-defined types.

The graphical notation represents data types in the form of trees: intermediate nodes represent existing abstract data types that are being instantiated, and their subtrees the actual parameters used in the instantiation.

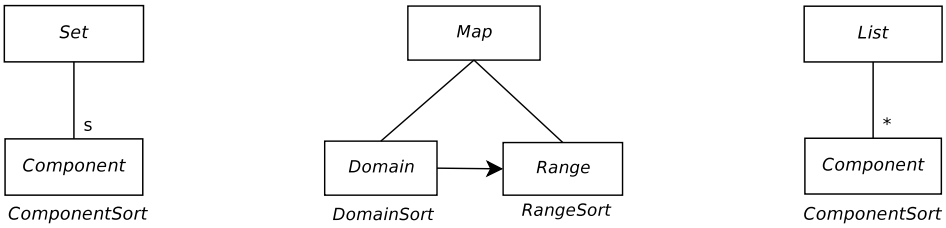
The leaves represent either primitive types or non-primitive sorts defined in other ATO's. A label under the corresponding leaf indicates its sort. When no such label is present under a leaf node, an external sort with the same name as the node is assumed.

3.2.1 Built-in data types representation and instantiation

Primitive data types ($Integer$, $Real$, $String$, $Boolean$, $Char$, etc) are represented by single boxes. They are always the leaves in the instantiation of composite data types.

Composite data types are represented by trees with a specific layout for each data type. Follows the representation used in Algebraic Prosoft for the built-in composite abstract data types. Those built-in composite data types are described, highlighting their graphical representation.

In each composite data type description, we list the sorts generated by the instantiation of that data type (**sorts**), as well as the parameter sorts needed for the instantiation (**formal sorts**).



(a) Sets graphical representation

(b) Maps graphical representation

(c) Lists graphical representation

Figure 3.2: Set, Map and List representations

3.2.1.1 Set

This is the abstract data type for sets. The most common set operations are available, as well as membership operations. As seen in figure 3.2(a), sets are represented by a single-child tree, where there is a small "s" above the child node.

- **Sorts:** Set.
- **Formal sorts:** $Component$.

3.2.1.2 Map

Map is the type for function-like objects, where for each input value (called "key"), there is one and only one associated object (that can be itself a set or list, but is still unique regarding that key). As seen in figure 3.2(b), maps are represented by a Map node, and

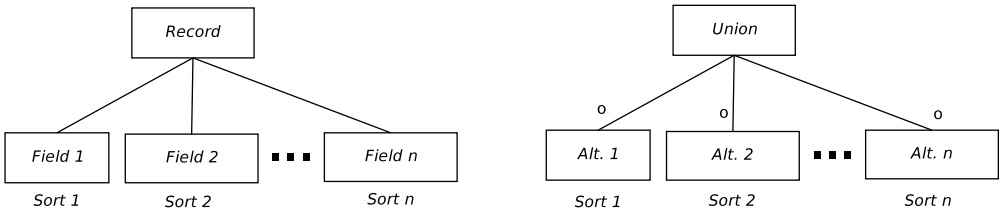
two child nodes linked by an arrow: the left one for the $Domain$ (values used as keys), and the right one for the $Range$.

- **Sorts:** Map.
- **Formal sorts:** $Domain$, $Range$.

3.2.1.3 $List$

$List$ is the type for sequences of objects. Differently from Set, in a $List$ duplicated elements are allowed, and the order of the elements is relevant. List is represented as in figure 3.2(c).

- **Sorts:** $List$.
- **Formal sorts:** $Component$.



(a) Records graphical representation

(b) Unions graphical representation

Figure 3.3: Record and Union representations

3.2.1.4 $Record$

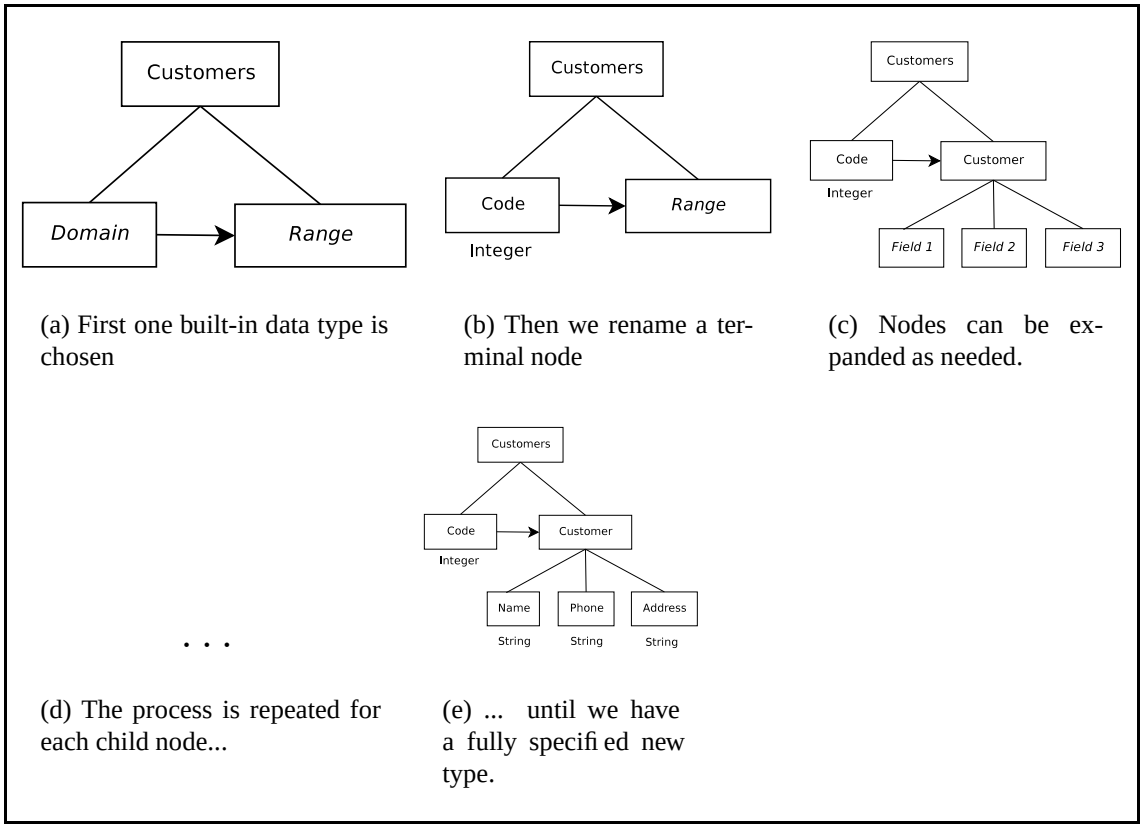
A $Record$ is a composite, non-homogeneous type. Each field is declared as being of its own sort. In the case of records, the name of each field (the name inside the box, see figure 3.3(a)) is used in the name of the corresponding operation that fetches the value of that field in a term from sort $Record$ (see section B in appendix).

- **Sorts:** $Record$.
- **Formal sorts:** $Field_1, Field_2, \ldots, Field_n$.

3.2.1.5 $Union$

$Union$ represent the union of n types, each prefixed with a tag. A term from $Union$ is either a term from $Sort_1$ tagged with Alt_1, or a term from $Sort_2$ tagged with Alt_2, ..., or a term from $Sort_n$ tagged with Alt_n. In Prosoft, the node names (see figure 3.3(b)) are used as the tags. The alternatives' names are used to build constructor operations for each one of the alternatives (see section B in appendix).

- **Sorts:** $Union$.
- **Formal sorts:** $Sort_1, Sort_2, \ldots, Sort_n$.



(a) First one built-in data type is chosen

(b) Then we rename a terminal node

(c) Nodes can be expanded as needed.

(d) The process is repeated for each child node...

(e) ... until we have a fully specified new type.

Figure 3.4: User-defined type: instantiation explained in a few steps.

3.2.2 Instantiating built-in types in user-defined data types

The creation of user-defined types is supported through instantiation of one or more built-in types. Users can create types that are lists of sets of maps, or maps from Integers to sets of records, and so on.

In general, a user-defined type is created following the criteria below:

a) Choose a built-in type and rename the main sort (the root of the tree) to reflect the new type. (figure 3.4(a))

b) For each child node (formal parameters):

- Rename its box according to the new type. (figure 3.4(a))
- If the node is a primitive or a sort defined elsewhere, specify as a label below it which sort it represents (alternatively the label can be blank to refer to a sort with the same name as the node). (figure 3.4(b))
- If the node must be itself a composite type, just “create” the corresponding tree under it and proceed to b). (figure 3.4(c))

3.2.3 Semantics’ informal notions

Suppose a built-in type $Type_1$, specified in an ATO named ATO_1. If we instantiate $Type_1$ as $Type_2$ inside a new specification ATO_2, using $Sort_i$ for $FormalSort_i$, this instantiation makes every operation (and its associated equations) in ATO_1 be inherited by ATO_2 under the following rules:

- Every occurrence of $Type_1$ in the operation's signature is replaced by $Type_2$.
- Every occurrence of $FormalSort_i$ in the operations signature is replaced by $Sort_i$.

3.3 Term Reduction and the ICS

In Algebraic Prosoft reduction can occur in two contexts: **local reduction**, where both the operations defined by the user and the operations inherited from the instantiated built-in types (as seen in section 3.2) are available; and **ICS reduction**, for references to external operations through the ICS operation.

Local reduction is used whenever an ATO specifies its equations in terms of local operations:

- equations are tested from top to bottom, for matching the term being reduced against each equation's left-hand side.
- when the term being reduced matches equation i's left-hand side, the term is replaced by the equation's right-hand side, replacing variables by the correspondent sub-terms in the original term, according to the matching. Then this new term is reduced again.
- If no equation matches the term being reduced, we then try to reduce each one of its sub-terms. If at least one is reduced, then we try to reduce the whole term again. Otherwise the reduction stops.

Within ICS, reduction is somewhat more complicated. The semantics of the application $ICS(ATO_i, op, \langle t_1, t_2, \ldots, t_n \rangle)$, with op defined as $op : E_1, E_2, \ldots, E_n \rightarrow E$, is as follows:

- apply operation op, defined in ATO_i, to the term t_1 (in other words, reduce $op(t_1)$ in the context of ATO_i)
- when $n > 1$:
  the op application's result **must** have the form $ICS(ATO_j, op')$, where $op' : E_2, \ldots, E_n \rightarrow E$; in this case, apply $ICS(ATO_j, op', \langle t_2, \ldots, t_n \rangle)$. Optionally, there can be intermediate results in terms of local operations, which should eventually reduce to something in the form $ICS(ATO_j, op')$.
- when $n = 1$:
  op application's result **must** be: **either** a term t from sort E, in this case the final result; **or** a term in the form $ICS(ATO_E, op'', \langle t'_1, t'_2, \ldots, t'_m \rangle)$, which is then reduced by this same process.

This semantics notion highlights one strong characteristics for ICS reduction: as seen in section 3.1, operations reduced by ICS must be monadic. This restriction limits what operations can be "called" through ICS.

Operations, however, *don't need* to be monadic. For example, most of operations in built-in data types that manipulate or operate objects won't be monadic: they can still be used in user-specified ATO's that include those built-in ATO's.

The ICS behavior can be illustrated by figure 3.5: each ICS call triggers a "jump" to another ATO to proceed reduction in that context. Is some sense, ICS acts like an *system bus* through which ATO's can exchange messages, as illustrated in figure 3.6.

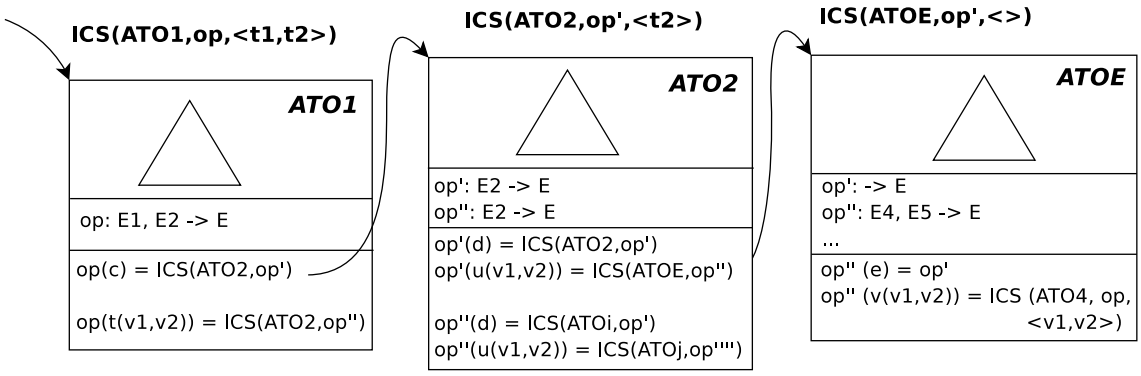


Figure 3.5: ATO's and ICS: ICS reduction in Prosoft

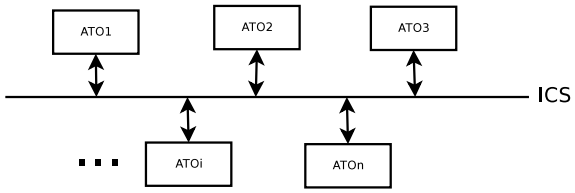


Figure 3.6: The ICS acts as a message bus between the ATO's

3.4 Final remarks

This chapter presented an overview of Algebraic Prosoft: how it can be used to model real-world problems, how to use its graphical notation in the definition of data types in Prosoft, as well as an informal notion of the semantics for the graphical notation (the instantiation of data types) and term reduction.

Next chapter presents Algebraic Prosoft's formal semantics.

4 PROSOFT FORMAL SEMANTICS

This chapter presents Algebraic Prosoft's formal semantics, using the denotational method as presented by Watt (WATT, 1991). Section 4.1 presents Algebraic Prosoft's syntax; section 4.2 relates the graphical representation of ATO's to their corresponding syntactic form in Algebraic Prosoft's syntax. section 4.3 introduces the notation used in the semantics definition. section 4.4 states the semantic domains used in the semantics; section 4.5 presents the core parts of the semantics; section 4.6 presents the semantics for instantiation of ATO's; section 4.7 presents the semantics for term matching; section 4.8 presents the auxiliary functions used in the earlier parts of semantics; section 4.9 summarizes the chapter.

4.1 Syntax

Most of semantics definitions rely on an abstract syntax for describing the language, but for Algebraic Prosoft we'll present its concrete semantics. Algebraic Prosoft's syntax is already very simple and easy to understand, and making it simpler would prejudice readability.

This way, presenting the actual syntax used in specifications won't distract the reader from what really matters here, the semantics definitions.

$Specification ::=$
  specification Id
    $Includes$
    $FormalSorts$
    $Sorts$
    $Operations$
    $Variables$
    $Equations$
  end

$Includes ::= IncludeDecl^*$

$IncludeDecl ::=$ **include** Id^*
  | **include instantiation of** Id
  | $RenameFormalSpec^*$
  | **renamed using** Id **for** Id

$RenameFormalSpec ::=$ **using** Id **for** Id

$FormalSorts ::= \varepsilon$
| **formal sort** Id
| **formal sorts** Id^*

$Sorts ::= \varepsilon$
| **sort** Id
| **sorts** Id^*

$Operations ::= \varepsilon$
| **operations** $Operation^*$

$Operation ::= Id : SortName^* \rightarrow SortName$

$SortName ::= Id$
| $ExternalSort\ Id\ Id$

$Variables ::= \varepsilon$
| **variables** $Variable^*$

$Variable ::= Id : SortName$

$Equations ::= \varepsilon$
| **equations** $Equation^*$

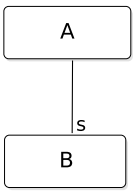
$Equation ::= Term$ = $Term$
| $Term$ = $Term$ **if** $Term$

$Term ::= Id$
| $Id\ ?$
| $Id\,(Term^*)$
| **ICS** (Id, Id)
| **ICS** $(Id, Id, [Term^*])$
| **if** $Term$ **then** $Term$ **else** $Term$

4.2 Relating graphical representation of ATO's and syntax

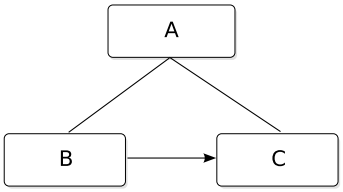
Every predefined ATO together with its graphical representation (as shown in section 3.2) has a corresponding syntax. This section associates the trees in their graphical representation with their corresponding syntactic form in the syntax used for semantics definitions, aiming to help the development of a future graphical tool for ATO's design and its integration with the presented semantics.

4.2.1 Sets



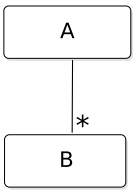

```
include instantiation of Sets
  using B for Element
renamed using A for Set
```

4.2.2 Maps



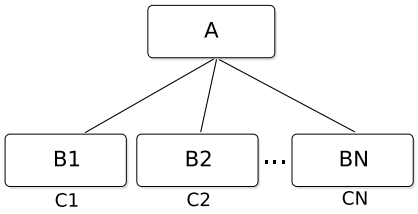

```
include instantiation of Maps
  using B for Domain,
  using C for Range
renamed using A for Map
```

4.2.3 Lists




```
include instantiation of Lists
  using B for Component
renamed using A for List
```

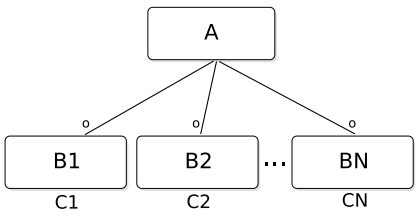
4.2.4 Records




```
include instantiation of Records_N
  using B1 for Field_1,
  using C1 for Sort_1,

  using B2 for Field_2,
  using C2 for Sort_2,
  ...
  using BN for Field_N,
  using CN for Sort_N
renamed using A for Record_N
```

4.2.5 Unions

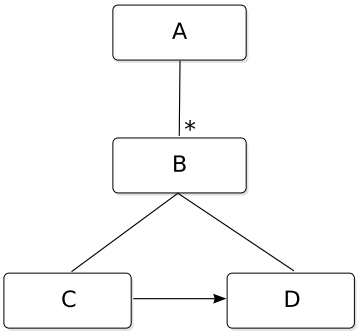



```
include instantiation of Unions_N
  using B1 for Tag_1,
  using C1 for Sort_1,

  using B2 for Tag_2,
  using C2 for Sort_2,
  ...
  using BN for Tag_N,
  using CN for Sort_N
renamed using A for Union_N
```

4.2.6 Multi-level trees and their instantiation

In the case of multi-level instantiation (e.g. sets of records, or lists of maps), the corresponding syntactic form must be created by sequencing the instantiations of each subtree in a bottom–up fashion. There is an example below:




```
include instantiation of Maps
  using C for Domain,
  using D for Range
renamed using B for Map

include instantiation of Lists
  using B for Component
renamed using A for List
```

4.3 Notation

In this section, we introduce the notation used in Algebraic Prosoft's semantics definition. At principle, the notation is the one used by (WATT, 1991) and (SCHMIDT, 1986), with some additions.

Since we'll be using $=$ for definition of variables, we'll represent equality as $\equiv$. This way, $x = y$ is read "x is defined as y". $x \equiv y$ reads as "x equals to y".

Other notation worth to mention is the representation of lists:

- A^* represents the domain of lists whose elements are of type A.
- $x \bullet xs$ represents a list that has x as its first element, and the list xs as its remainder.
- $[x]$ represents a list that consists of only one element x.
- $+\!\!+$ is the lists (and strings) concatenation operator.

Some functions on lists are commonly used: they are map, $foldl$, $init$ and $last$.

$$
\begin{aligned}
&map : (A \to B) \to A^* \to B^* \\
&map\ f\ [\,] = [\,] \\
&map\ f\ (x \bullet xs) = (f\ x) \bullet (map\ f\ xs)
\end{aligned}
$$

map applies a function f to every element of a list and yields another list with the results of the applications. Example: $map\ (+1)\ [1,2,3] = [2,3,4]$

$$
\begin{aligned}
&foldl : (A \to B \to A) \to A \to B^* \to A \\
&foldl\ f\ z\ [\,] = z \\
&foldl\ f\ z\ (x \bullet xs) = foldl\ f\ (f\ z\ x)\ xs
\end{aligned}
$$

$foldl$ operates z with the first element of a list, through the function f. This is repeated with the result of the previous operation and the rest of the list, until its end. For example: $foldl\ (+\!\!+)\ \texttt{"a"}\ [\texttt{"b"}, \texttt{"c"}] = \texttt{"abc"}$.

$init$ yields all the elements of a list, except for the last one.

$$
\begin{aligned}
&init : A^* \to A^* \\
&init\ [x] = [\,] \\
&init\ (x \bullet xs) = x \bullet (init\ xs)
\end{aligned}
$$

$last$ yields the last element of a list.

$$
\begin{aligned}
&last : A^* \to A \\
&last\ [x] = x \\
&last\ (x \bullet xs) = last\ xs
\end{aligned}
$$

Another domain largely used are maps. $A \xrightarrow{m} B$ is the domain of maps from A to B. If m is a map, $m\,(x)$ is the value at the key x. $\{\}$ is the empty map. $m \underline{\cup} \{x \mapsto y\}$ represents a map that is just like m, except for mapping x to y. In general:

$$(a \underline{\cup} b)(x) = \begin{cases} b(x),\ if\ x \in domain(b) \\ a(x),\ otherwise \end{cases}$$

Syntactic elements are represented inside double brackets, like $[\![$**if** t_1 **then** t_2 **else** $t_3]\!]$. Sets and functions use the regular notation.

4.4 Semantic domains

The *Prosoft* domain represents instances of the Prosoft Environment: they hold all available ATO's, indexed by their names. *Prosoft* is a mapping from identifiers to ATO's.

$$Prosoft = Id \xrightarrow{m} ATO$$

We represent an ATO by a triple with its operations, variables and equations. ATO is a Cartesian-product domain, in which each element is a list of, respectively, operations, variables, and equations.

$$ATO = Operation^* \times Variable^* \times Equation^*$$

4.5 Semantic functions

specify is the semantic function for a series of ATO's specifications. It takes the sequence of specifications and produces an environment of ATO's, in which we can search ATO's by their names.

Since we expect that specifications presented later override earlier ones, recursion is done from the end of the specifications sequence instead of being done from the beginning.

$$\begin{array}{l} specify : Specification^* \to Prosoft \\ specify\ specs = specify'\ specs\ specs\ \textbf{where} \\ \quad specify' : Specification^* \to Specification^* \to Prosoft \\ \quad specify'\ \_\ [\,] = \{\} \\ \quad specify'\ allSpecs\ specs = \\ \quad\quad (specify'\ allSpecs\ (init\ specs)) \underline{\cup} \{id \mapsto ato\} \\ \quad\quad\quad \textbf{where} \left[\!\!\left[\begin{array}{l} \textbf{specification}\ id \\ \quad \_ \\ \quad \_ \\ \quad \_ \\ \quad \_ \\ \quad \_ \\ \quad \_ \\ \textbf{end} \end{array} \right]\!\!\right] = (last\ specs) \\ \quad\quad\quad\quad ato = instantiate\ allSpecs\ noRename\ (last\ specs) \end{array}$$

reduce is the generic reduction semantic function.

$$reduce : Term \to ATO \to Prosoft \to Term$$

The definition of $reduce$ is based on cases, one for each type of term.
In the case of a ICS call, $reduce$ dispatches to the ics semantic function.

$$
\begin{array}{l}
reduce\ [\![\textbf{ICS}\,(atoname, op, [args])]\!]\ \_\ env = \\
\quad ics\ atoname\ op\ args\ env
\end{array}
$$

For the other form of ICS (operation reference) $[\![\textbf{ICS}\,(atoname, op)]\!]$, we just try to reduce the term op in the context of the ATO $atoname$. In the case when there is no ATO with that name, we don't reduce.

$$
\begin{array}{l}
reduce\ [\![\textbf{ICS}\,(atoname, op)]\!]\ \_\ env = \\
\quad \textbf{case}\ env\,(atoname)\ \textbf{of} \\
\qquad \bot \rightarrow [\![\textbf{ICS}\,(atoname, op)]\!] \\
\qquad ato \rightarrow \textbf{let}\ (\_, \_, eqs) = ato \\
\qquad\qquad\quad \textbf{in}\ reduce\ [\![op]\!]\ ato\ env
\end{array}
$$

The next $reduce$ cases represent *local reduction*.
The reduction of terms in the form $[\![\textbf{if}\ cond\ \textbf{then}\ t_1\ \textbf{else}\ t_2]\!]$ consists in first reducing $cond$. If $cond$ reduces to $[\![\texttt{"true"}]\!]$, then we yield the reduction of t_1. If $cond$ reduces to $[\![\texttt{"false"}]\!]$, then we yield the reduction of t_2. If $cond$ reduces to something else, we yield a partially-reduced term. [1]

$$
\begin{array}{l}
reduce\ [\![\textbf{if}\ cond\ \textbf{then}\ t_1\ \textbf{else}\ t_2]\!]\ ato\ env = \\
\quad \textbf{let}\ cond' = reduce\ cond\ ato\ env \\
\qquad\ t_1' = reduce\ t_1\ ato\ env \\
\qquad\ t_2' = reduce\ t_2\ ato\ env \\
\quad \textbf{in if}\ (cond' \equiv [\![\texttt{"true"}]\!]) \\
\qquad \textbf{then}\ t_1' \\
\qquad \textbf{else if}\ (cond' \equiv [\![\texttt{"false"}]\!]) \\
\qquad\qquad \textbf{then}\ t_2' \\
\qquad\qquad \textbf{else}\ [\![\textbf{if}\ cond'\ \textbf{then}\ t_1'\ \textbf{else}\ t_2']\!]
\end{array}
$$

All other types of term are reduced by $reduce'$, which iterates on the equations defined in the ATO ato looking for a match (see section 4.7 for term matching).

$$
reduce\ t\ \overbrace{(\_, \_, eqs)}^{ato}\ \ env = reduce'\ ato\ eqs\ eqs\ env\ t
$$

ics, the semantic function for ICS reduction, is as follows:

$$
\begin{array}{l}
ics : String \rightarrow String \rightarrow Term^* \rightarrow Prosoft \rightarrow Term \\
ics\ ato\ op\ (t \bullet ts)\ env = \\
\quad \textbf{let}\ found = env\,(ato) \\
\quad \textbf{in case}\ found\ \textbf{of} \\
\qquad \bot \rightarrow [\![\textbf{ICS}\,(ato, op, [(t \bullet ts)])]\!] \\
\qquad ato \rightarrow \textbf{let}\ term = [\![op\,([t])]\!] \\
\qquad\qquad\qquad\quad result = reduce\ term\ ato\ env \\
\qquad\qquad\quad \textbf{in if}\ (ts \equiv [\,])
\end{array}
$$

[1] Note that as we are using lambda calculus, no assumption is made regarding the actual order of evaluation of $cond'$, t_1' and t_2'.

$$
\begin{array}{l}
\quad\textbf{then case } result \textbf{ of} \\
\qquad\quad [\![\textbf{ICS}\,(ato', op', [ts'])]\!] \rightarrow ics\ ato'\ op'\ ts'\ env \\
\qquad\quad \_ \rightarrow result \\
\textbf{else let } [\![\textbf{ICS}\,(ato', op')]\!] = result \\
\qquad \textbf{in } ics\ ato'\ op'\ ts\ env
\end{array}
$$

The $reduce'$ auxiliary function receives the equations twice, in order to keep both the full equations list and the list of remaining equations available in the reduction try.

$$reduce' : ATO \rightarrow Equation^* \rightarrow Equation^* \rightarrow Prosoft \rightarrow Term \rightarrow Term$$

When there are no more equations to try, atoms reduce to themselves:

$$reduce'\ ato\ eqs\ [\,]\ env\ \overbrace{[\![id]\!]}^{t} = t$$

Input variables are always reduced to themselves (that is, they are not reduced at all)[2].

$$reduce'\ \_\ \_\ \_\ \_\ \overbrace{[\![id\ ?]\!]}^{t} = t$$

When applying an operator over some terms and there are no more equations to try (because all others were already tried), maybe we can first reduce every argument, and then try to reduce the whole term. When none of the subterms reduce, we stop the reduction.

$$
\begin{array}{l}
reduce'\ ato\ eqs\ [\,]\ env\ \overbrace{[\![op\,(args)]\!]}^{t} = \\
\quad \textbf{let } rargs = map\ (\lambda\ t.\ reduce\ t\ ato\ env)\ args \\
\quad \textbf{in if } args \not\equiv rargs \\
\qquad \textbf{then } reduce\ [\![op\,(rargs)]\!]\ ato\ env \\
\qquad \textbf{else } t
\end{array}
$$

For direct equations in the form $t_1 = t_2$, we just try to match the term t with t_1, producing the term environment env'. If this matching succeeds, we then reduce t do t_2 (substituted by env'). Otherwise we try $reduce'$ with the next equation.

$$
\begin{array}{l}
reduce'\ ato\ eqs\ ([\![t_1 = t_2]\!] \bullet eqsR)\ env\ t = \\
\quad \textbf{let } (tv, env') = match\ t_1\ t\ ato\ \{\} \\
\quad \textbf{in if } tv \\
\qquad \textbf{then } reduce\ (subs\ t_2\ env')\ ato\ env \\
\qquad \textbf{else } reduce'\ ato\ eqs\ eqsR\ env\ t
\end{array}
$$

For equations in the form $[\![t_1 = t_2 \textbf{ if } cond]\!]$, we first try to match the given term against t_1, producing the term environment env. If that succeeds, we then reduce $cond$ substituted by env. If $cond$ reduces to $true$, then we reduce t to t_2 substituted with env. Otherwise we continue the reduction, trying $reduce'$ with the next/ equation.

[2]Transforming input variables into actual terms is left to the user interface, and not considered as part of the semantics for Algebraic Prosoft.

$$
\begin{array}{l}
\mathit{reduce'}\ \mathit{ato}\ \mathit{eqs}\ (\llbracket t_1 = t_2\ \textbf{if}\ \mathit{cond} \rrbracket \bullet \mathit{eqsR})\ \mathit{env}\ t = \\
\quad \textbf{let}\ (\mathit{matches}, \mathit{env'}) = \mathit{match}\ t_1\ t\ \mathit{ato}\ \{\} \\
\quad \textbf{in if}\ (\mathit{matches} \wedge \\
\qquad\quad (\mathit{reduce}\ (\mathit{subs}\ \mathit{cond}\ \mathit{env'})\ \mathit{ato}\ \mathit{env} \equiv \llbracket \texttt{"true"} \rrbracket) \\
\qquad)\ \\
\qquad \textbf{then}\ \mathit{reduce}\ (\mathit{subs}\ t_2\ \mathit{env'})\ \mathit{ato}\ \mathit{env} \\
\qquad \textbf{else}\ \mathit{reduce'}\ \mathit{ato}\ \mathit{eqs}\ \mathit{eqsR}\ \mathit{env}\ t
\end{array}
$$

The substitution function $subs$ substitutes the variables in a term by the corresponding term in a term environment, and is used when there is a match of a term against some equation's left-hand side. The variables in the left-hand side that matches subterms in the reducing term are put in a term environment, and the equation's right-hand side gets its variables substituted with that environment.

$subs$ is defined as following:

$$
\begin{array}{l}
\mathit{subs} : \mathit{Term} \rightarrow (\mathit{Id} \xrightarrow{m} \mathit{Term}) \rightarrow \mathit{Term} \\
\mathit{subs}\ \llbracket \mathit{id} \rrbracket\ \mathit{env} = \mathit{findTerm}\ \mathit{env}\ \mathit{id} \\
\mathit{subs}\ \llbracket \mathit{op}\ (\mathit{args}) \rrbracket\ \mathit{env} = \llbracket \mathit{op}\ ((\mathit{map}\ (\lambda\ t.\ \mathit{subs}\ t\ \mathit{env})\ \mathit{args})) \rrbracket \\
\mathit{subs}\ \llbracket \textbf{if}\ \mathit{cond}\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \rrbracket\ \mathit{env} = \\
\quad \llbracket \textbf{if}\ (\mathit{subs}\ \mathit{cond}\ \mathit{env})\ \textbf{then}\ (\mathit{subs}\ t_1\ \mathit{env})\ \textbf{else}\ (\mathit{subs}\ t_2\ \mathit{env}) \rrbracket \\
\mathit{subs}\ \llbracket \textbf{ICS}\ (\mathit{ato}, \mathit{op}, [\mathit{terms}]) \rrbracket\ \mathit{env} = \\
\quad \llbracket \textbf{ICS}\ (\mathit{ato}, \mathit{op}, [(\mathit{map}\ (\lambda\ x.\ \mathit{subs}\ x\ \mathit{env})\ \mathit{terms})]) \rrbracket \\
\mathit{subs}\ \overbrace{\llbracket \textbf{ICS}\ (\_, \_) \rrbracket}^{t}\ \ \_ = t \\
\mathit{subs}\ \overbrace{\llbracket \_\ ? \rrbracket}^{t}\ \ \_ = t
\end{array}
$$

The $findTerm$ function yields the term corresponding to id in the term environment env, if there is such term in env. Otherwise it yields id itself.

$$
\begin{array}{l}
\mathit{findTerm} : (\mathit{Id} \xrightarrow{m} \mathit{Term}) \rightarrow \mathit{Id} \rightarrow \mathit{Term} \\
\mathit{findTerm}\ \mathit{env}\ \mathit{id} = \\
\quad \textbf{case}\ (\mathit{env}\ (\mathit{id}))\ \textbf{of} \\
\qquad \bot \rightarrow \llbracket \mathit{id} \rrbracket \\
\qquad x \rightarrow x
\end{array}
$$

4.6 Instantiation

Instantiation if the process of creating ATO's from specifications. It consists of collecting its operations, variables and equations.

A special case is when an ATO instantiates another ATO. This normally happens when an ATO (say, a_1) includes an instantiation of another ATO (say, a_2), where a_2 has formal sort parameters. All the operations present in a_2 are included in a_1, with the formal sort parameters replaced by the actual sort parameters. This is the case when the built-in ATO's that define composite abstract data types are instantiated in user-defined ATO's.

In this section we develop the semantics for ATO instantiation.

First, we define a function domain for renaming functions. Renaming functions take a $SortName$ and supply another $SortName$, supposed to substitute the first one in the specification.

$$Renaming = SortName \to SortName$$

$noRename$ is the identity renaming function. It returns just the same argument that it receives.

$$noRename = \lambda\, x.\, x$$

$rename$ creates a new renaming function based on another one, adding a new renaming pair.

$$\begin{array}{l} rename : Renaming \to SortName \to SortName \to Renaming \\ rename\ r\ old\ new = \lambda\, x.\ \textbf{if}\ x \equiv old\ \textbf{then}\ new\ \textbf{else}\ r\ x \end{array}$$

The $instantiate$ function builds an ATO from its textual specification, adding all the included specifications from their respective specification.

It takes the whole list of available specifications, a renaming function and the specification to be instantiated.

$$\begin{array}{l} instantiate : Specification^* \to Renaming \to Specification \to ATO \\ instantiate\ specs\ renaming\ spec = \\ \quad \textbf{let}\ included = foldl\ (+\!\!+)\ [\,]\ (map\ (\lambda\, i.\ include\ i\ specs)\ incs) \\ \qquad included\_as\_ato = joinATOs\ included \\ \qquad (ops', vars', eqs') = included\_as\_ato \\ \quad \textbf{in}\ (map\ (instantiateOperation\ renaming)\ (getOperations\ ops) +\!\!+ ops', \\ \qquad map\ (instantiateVariable\ renaming)\ (getVariables\ vars) +\!\!+ vars', \\ \qquad map\ (instantiateEquation\ renaming)\ (getEquations\ eqs) +\!\!+ eqs' \\ \qquad) \\ \quad \textbf{where}\ \left[\!\!\left[\begin{array}{l} \textbf{specification}\ \_ \\ \quad incs \\ \quad \_ \\ \quad \_ \\ \quad ops \\ \quad vars \\ \quad eqs \\ \textbf{end} \end{array} \right]\!\!\right] = spec \end{array}$$

The $getOperations$, $getVariables$ and $getEquations$ functions build lists of their respective entities from their syntactic representation.

$$\begin{array}{l} getOperations : Operations \to Operation^* \\ getOperations\ [\![\varepsilon]\!] = [\,] \\ getOperations\ [\![\textbf{operations}\ ops]\!] = ops \end{array}$$

$$\begin{array}{l} getVariables : Variables \to Variable^* \\ getVariables\ [\![\varepsilon]\!] = [\,] \\ getVariables\ [\![\textbf{variables}\ vars]\!] = vars \end{array}$$

$$\begin{array}{l} getEquations : Equations \to Equation^* \\ getEquations\ [\![\varepsilon]\!] = [\,] \\ getEquations\ [\![\textbf{equations}\ eqs]\!] = eqs \end{array}$$

The $joinATOs$ function simply does the "fusion" of a list of ATO's in just one, concatenating operations, variables and equations lists.

$$
\begin{array}{l}
joinATOs : ATO^* \to ATO \\
joinATOs\ [\,] = ([\,], [\,], [\,]) \\
joinATOs\ ((o_1, v_1, e_1) \bullet as) = \\
\quad \textbf{let}\ (o_2, v_2, e_2) = joinATOs\ as \\
\quad \textbf{in}\ (o_1 +\!\!+ o_2, v_1 +\!\!+ v_2, e_1 +\!\!+ e_2)
\end{array}
$$

Given an include declaration in one specification, the $include$ function instantiates the referenced ATO's:

$$include : IncludeDecl \to Specification^* \to ATO^*$$

In the case of the $[\![$**include** $ids]\!]$ declaration, we simply instantiate all the referenced ATO's, without any renaming.

$$
\begin{array}{l}
include\ [\![\textbf{include}\ ids]\!]\ specs = \\
\quad \textbf{let}\ referenced = map\ (findSpec\ specs)\ ids \\
\quad \textbf{in}\ map\ (\lambda\ spec.\ instantiate\ specs\ noRename\ spec) \\
\qquad\qquad referenced
\end{array}
$$

In the case of an actual instantiation, we must build a renaming function, then use it to instantiate the referenced ATO (just one, in this case).

Since $Records$ and $Unions$ have a variable number of fields/choices, and depending on this number we have different specifications, when the instantiated specification is called `"Records"` or `"Unions"`, we need to create such specification dynamically using the functions $createRecordSpec$ and $createUnionSpec$. These functions are omitted: they yield unary lists (see $include$'s signature), whose element is an ATO corresponding to the desired specification. See section 6.3 for a deeper explanation of this issue.

$$
\begin{array}{l}
include\ \left[\!\!\left[\begin{array}{l}\textbf{include instantiation of}\ specname \\ \quad renameFormals \\ \textbf{renamed using}\ new\ \textbf{for}\ old\end{array}\right]\!\!\right]\ specs = \\
\quad \textbf{let}\ spec' = \textbf{case}\ specname\ \textbf{of} \\
\qquad\qquad \texttt{"Records"} \to (createRecordSpec\ renameFormals\ new\ old) \\
\qquad\qquad \texttt{"Unions"} \to (createUnionSpec\ renameFormals\ new\ old) \\
\qquad\qquad x \to findSpec\ specs\ specname \\
\quad \textbf{in case}\ spec'\ \textbf{of} \\
\qquad \bot \to \bot \\
\qquad spec \to \textbf{let}\ pairs = map\ renamePair\ renameFormals \\
\qquad\qquad\quad \textbf{in let}\ renaming = rename\ (pairsToRenaming\ pairs) \\
\qquad\qquad\qquad\qquad\qquad\qquad [\![old]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad [\![new]\!] \\
\qquad\qquad\quad \textbf{in}\ [instantiate\ specs\ renaming\ spec]
\end{array}
$$

The $instantiateOperation$, $instantiateVariable$, $instantiateEquation$ and $instantiateTerm$ functions apply the renaming to the corresponding elements, and are as follows:

$$
\begin{array}{l}
instantiateOperation : Renaming \rightarrow Operation \rightarrow Operation \\
instantiateOperation\ renaming\ [\![id : domain \rightarrow range]\!] = \\
\quad [\![id : (map\ renaming\ domain) \rightarrow (renaming\ range)]\!]
\end{array}
$$

$$
\begin{array}{l}
instantiateVariable : Renaming \rightarrow Variable \rightarrow Variable \\
instantiateVariable\ renaming\ [\![id : sort]\!] = [\![id : (renaming\ sort)]\!]
\end{array}
$$

$$
\begin{array}{l}
instantiateEquation : Renaming \rightarrow Equation \rightarrow Equation \\
instantiateEquation\ r\ [\![t_1 = t_2]\!] = \\
\quad [\![(instantiateTerm\ r\ t_1) = (instantiateTerm\ r\ t_2)]\!] \\
instantiateEquation\ r\ [\![t_1 = t_2\ \textbf{if}\ t_3]\!] = \\
\quad [\![(instantiateTerm\ r\ t_1) = (instantiateTerm\ r\ t_2)\ \textbf{if}\ (instantiateTerm\ r\ t_3)]\!]
\end{array}
$$

$$
\begin{array}{l}
instantiateTerm : Renaming \rightarrow Term \rightarrow Term \\
instantiateTerm\ r\ [\![\textbf{ICS}\,(ato, op)]\!] = \\
\quad [\![\textbf{ICS}\,((f\ (r\ [\![ato]\!])), op)]\!] \\
\quad\quad \textbf{where}\ f\ [\![new]\!] = new \\
instantiateTerm\ r\ [\![\textbf{ICS}\,(ato, op, [terms])]\!] = \\
\quad [\![\textbf{ICS}\,((f\ (r\ [\![ato]\!])), op, [(map\ (instantiateTerm\ r)\ terms)])]\!] \\
\quad\quad \textbf{where}\ f\ [\![new]\!] = new \\
instantiateTerm\ r\ [\![\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3]\!] = \\
\quad [\![\textbf{if}\ (instantiateTerm\ r\ t_1)\ \textbf{then}\ (instantiateTerm\ r\ t_2)\ \textbf{else}\ (instantiateTerm\ r\ t_3)]\!] \\
instantiateTerm\ \_\ t = t
\end{array}
$$

The $renamePair$ and $pairsToRenaming$ functions are reponsible for transforming the syntatic representation of renaming clauses into a renaming function.

$$
\begin{array}{l}
renamePair : RenameFormalSpec \rightarrow (SortName, SortName) \\
renamePair\ [\![\textbf{using}\ newname\ \textbf{for}\ oldname]\!] = \\
\quad ([\![newname]\!], [\![oldname]\!])
\end{array}
$$

$$
\begin{array}{l}
pairsToRenaming : [(SortName, SortName)] \rightarrow Renaming \\
pairsToRenaming\ [\,] = noRename \\
pairsToRenaming\ ps = \\
\quad \textbf{let}\ (new, old) = last\ ps \\
\quad \textbf{in}\ rename\ (pairsToRenaming\ (init\ ps))\ old\ new
\end{array}
$$

4.7 Matching

Term matching in Prosoft is always done in the context of one specific ATO, and respects the following rules for each type of term (regarding the abstract syntax):

1. Term in the forms $[\![\textbf{ICS}\,(ato, op)]\!]$ and $[\![\textbf{ICS}\,(ato, op, [terms])]\!]$ don't match with any other terms. In fact, matching of terms in those forms doesn't make any sense at all.

2. An atom $[\![i]\!]$ matches with a given term t under the following conditions:

   - If i is the name a 0-ary constructor in the current ATO (i.e., a constant), i matches with t if and only if $i \equiv t$, producing no bindings.

- If i is a variable in the current ATO, them there are two cases:
  (a) if the identifier i already matched with some term (say, t') before the current matching, then t must be the same as t'.
  (b) if the identifier i din't match with any other term, then it matches with any term t, producing a binding $\{i \mapsto t\}$.

3. An input variable $[\![i\ ?]\!]$ doesn't match with any other term. [3]

4. $op(t_1, t_2, \dots)$ matches with $op'(t'_1, t'_2, \dots)$ if:

   - $op \equiv op'$
   - $\forall i$, t_i matches with t'_i, generating a binding env_i.

   In this case, the binding generated in the matching is the combination of all the generated env_i bindings.

5. There is no other matching possibility.

That said, to match two terms and check if they match, we need:

1. the ATO in which the matching is occurring.
2. the currently produced bindings, so we can *(i)* check for previous matchings and *(ii)* add new bindings that may be produced.

Thus, the matching function has the following type:

$$match : Term \to Term \to ATO \to (Id \xrightarrow{m} Term) \to (Bool \times (Id \xrightarrow{m} Term))$$

The first applicable situation is for atoms $[\![id]\!]$:

$$\begin{array}{l} match\ \overbrace{[\![id]\!]}^{t}\ \ t'\ ato\ env = \\ \quad \textbf{if}\ isConstant\ ato\ id \\ \quad \textbf{then}\ (t \equiv t', env) \\ \quad \textbf{else case}\ (\mathit{findOperation}\ ato\ id)\ \textbf{of} \\ \qquad x \to (t \equiv t', env) \\ \qquad \bot \to \textbf{case}\ (env\ (id))\ \textbf{of} \\ \qquad\qquad\qquad term \to (t' \equiv term, env) \\ \qquad\qquad\qquad \bot \to (True, insert\ id\ t'\ env) \end{array}$$

The second applicable situation is for operators in the form $[\![op\ (terms)]\!]$:

$$\begin{array}{l} match\ [\![op_1\ (args_1)]\!]\ [\![op_2\ (args_2)]\!]\ ato\ env = \\ \quad \textbf{if}\ op_1 \equiv op_2 \\ \quad \textbf{then}\ (matchList\ args_1\ args_2\ ato\ env) \\ \quad \textbf{else}\ (False, env) \end{array}$$

[3] After being substituted by its actual input, however, the term for which $[\![i\ ?]\!]$ is a placeholder can match with some other term in further reduction steps.

There are no other possibilities for matching:

$$match\ \_\ \_\ \_\ env = (False, env)$$

The function $matchList$ determines recursively if the operator arguments match and combines the produced bindings. Is is defined as follows:

$$\begin{aligned}
&matchList : Term^* \to Term^* \to ATO \to (Id \xrightarrow{m} Term) \to \\
&\qquad (Bool \times (Id \xrightarrow{m} Term)) \\
&matchList\ (t \bullet ts)\ [\,]\ _\ env = (False, env) \\
&matchList\ [\,]\ [\,]\ _\ env = (True, env) \\
&matchList\ [\,]\ (t \bullet ts)\ _\ env = (False, env) \\
&matchList\ (t \bullet ts)\ (t' \bullet ts')\ ato\ env = \\
&\quad \textbf{let}\ (tv, env') = match\ t\ t'\ ato\ env \\
&\quad \textbf{in if}\ tv \\
&\qquad \textbf{then}\ matchList\ ts\ ts'\ ato\ env' \\
&\qquad \textbf{else}\ (False, env')
\end{aligned}$$

4.8 Auxiliary Functions

This section define auxiliary functions used in the other parts of Algebraic Prosoft's semantics definition. Some of them are only used in the prototype implementation, but were kept here in favor or some organization.

The $findOperation$ and $findVariable$ functions are helpers that encapsulate looking for, respectively, operations and variables by their names.

$$\begin{aligned}
&findOperation : ATO \to Id \to Operation_\perp \\
&findOperation\ (ops, _, _)\ id = \\
&\quad findOperation'\ ops\ id\ \textbf{where} \\
&\qquad findOperation' : Operation^* \to Id \to Operation_\perp \\
&\qquad findOperation'\ [\,]\ id = \perp \\
&\qquad findOperation'\ (op \bullet ops)\ id = \\
&\qquad\quad \textbf{let}\ [\![opid : _ \to _]\!] = op \\
&\qquad\quad \textbf{in if}\ opid \equiv id \\
&\qquad\qquad \textbf{then}\ op \\
&\qquad\qquad \textbf{else}\ findOperation'\ ops\ id
\end{aligned}$$

$$\begin{aligned}
&findVariable : ATO \to Id \to Variable_\perp \\
&findVariable\ (_, vars, _)\ id = \\
&\quad findVariable'\ vars\ id\ \textbf{where} \\
&\qquad findVariable' : Variable^* \to Id \to Variable_\perp \\
&\qquad findVariable'\ [\,]\ id = \perp \\
&\qquad findVariable'\ (var \bullet vars)\ id = \\
&\qquad\quad \textbf{let}\ [\![vid : _]\!] = var \\
&\qquad\quad \textbf{in if}\ vid \equiv id \\
&\qquad\qquad \textbf{then}\ var \\
&\qquad\qquad \textbf{else}\ findVariable'\ vars\ id
\end{aligned}$$

The $isConstructor$ function tells if id names a constructor operation in the ATO ato, i.e., if it has no definition among ATO's equations.

```
isConstructor : ATO → Id → Bool
isConstructor ato id =
  case findOperation ato id of
    ⊥ → False
    _ → let (_, _, eqs) = ato
        in isConstructor' eqs id where
          isConstructor' : Equation* → Id → Bool
          isConstructor' [] _ = True
          isConstructor' (eq • eqs) id =
            case eq of
              ⟦⟦id'⟧ = _⟧ → (id ≢ id' ∧ isConstructor' eqs id)
              ⟦⟦id' (_)⟧ = _⟧ → (id ≢ id' ∧ isConstructor' eqs id)
              ⟦⟦id'⟧ = _ if _⟧ → (id ≢ id' ∧ isConstructor' eqs id)
              ⟦⟦id' (_)⟧ = _ if _⟧ → (id ≢ id' ∧ isConstructor' eqs id)
              _ → isConstructor' eqs id
```

The *isConstant* function tells if *id* is the name of a constant in a given ATO *ato*.

```
isConstant : ATO → Id → Bool
isConstant ato id =
  case (findOperation ato id) of
    ⊥ → False
    ⟦id : range → domain⟧ → ((range ≡ []) ∧ isConstructor ato id)
```

The *join* function takes a string *sep* and a list of strings *ss*, and yields an string with all elements of *ss* concatenated, interspersed by *sep*.

```
join : String → String* → String
join sep ss = foldl (++) "" (intersperse sep ss)
```

The *findSpec* auxiliary function just finds a specification by its id among all specifications:

```
findSpec : Specification* → Id → Specification⊥
findSpec [] id = ⊥
findSpec (s • ss) id =
  let ⟦| specification theId
         _
         _
         _
         _
         _
         _
       end |⟧ = s
  in if theId ≡ id
     then s
     else findSpec ss id
```

The *getInputs* function takes a term and returns the set of input variables (⟦*x* ?⟧) present in that term.

$$
\begin{aligned}
&getInputs : Term \to \mathcal{P}(String)\\
&getInputs\ [\![\_]\!] = \emptyset\\
&getInputs\ [\![\textbf{ICS}\,(\_,\_)]\!] = \emptyset\\
&getInputs\ [\![\_\,(terms)]\!] =\\
&\quad foldl\ (\lambda\ a.\ \lambda\ b.\ a \cup b)\ \emptyset\ (map\ getInputs\ terms)\\
&getInputs\ [\![\textbf{ICS}\,(\_,\_,[terms])]\!] =\\
&\quad foldl\ (\lambda\ a.\ \lambda\ b.\ a \cup b)\ \emptyset\ (map\ getInputs\ terms)\\
&getInputs\ [\![id\ ?]\!] = \{id\}\\
&getInputs\ [\![\textbf{if}\ cond\ \textbf{then}\ t_1\ \textbf{else}\ t_2]\!] =\\
&\quad foldl\ (\lambda\ a.\ \lambda\ b.\ a \cup b)\ \emptyset\ (map\ getInputs\ [cond, t_1, t_2])
\end{aligned}
$$

The $insertInputs$ functions takes a mapping $inputs$, from identifiers to terms, and a term t, and replaces all the input variables in t by the corresponding term in $inputs$.

$$
\begin{aligned}
&insertInputs : (String \xrightarrow{m} Term) \to Term \to Term\\
&insertInputs\ inputs\ \overbrace{[\![id\ ?]\!]}^{t} =\\
&\quad \textbf{case}\ (inputs\,(id))\ \textbf{of}\\
&\qquad \bot \to t\\
&\qquad term \to term\\
&insertInputs\ inputs\ [\![op\,(terms)]\!] =\\
&\quad [\![op\,((map\ (insertInputs\ inputs)\ terms))]\!]\\
&insertInputs\ inputs\ [\![\textbf{ICS}\,(ato, op, [terms])]\!] =\\
&\quad [\![\textbf{ICS}\,(ato, op, [(map\ (insertInputs\ inputs)\ terms)])]\!]\\
&insertInputs\ inputs\ [\![\textbf{if}\ cond\ \textbf{then}\ t_1\ \textbf{else}\ t_2]\!] = [\![\textbf{if}\ cond'\ \textbf{then}\ t_1'\ \textbf{else}\ t_2']\!]\\
&\quad \textbf{where}\ cond' = insertInputs\ inputs\ cond\\
&\qquad\qquad t_1' = insertInputs\ inputs\ t_1\\
&\qquad\qquad t_2' = insertInputs\ inputs\ t_2\\
&insertInputs\ \_\ \overbrace{[\![\_]\!]}^{t} = t\\
&insertInputs\ \_\ \overbrace{[\![\textbf{ICS}\,(\_,\_)]\!]}^{t} = t
\end{aligned}
$$

4.9 Final remarks

This chapter presented Algebraic Prosoft's formal semantics, using the denotational method.

Algebraic Prosoft's syntax was presented, together with a relation between the graphical representation of ATO's and their corresponding declarations in the presented syntax. Algebraic Prosoft's semantics was presented, divided in distinct sections for the core semantics (the semantics of term reduction), instantiation (the semantics of ATO's instantiation), matching (the semantics of term matching). Finally, all the used auxiliary functions were presented.

Next chapter presents research work done in semantic prototyping with the Haskell programming language, which guided Algebraic Prosoft's prototyping presented in chapter 6.

5 SEMANTICS-BASED LANGUAGE PROTOTYPING WITH HASKELL

This chapter presents the result of research done in semantic prototyping with the Haskell programming language.

5.1 Introduction

In software projects, the later an error is discovered, the higher is the cost for fixing it (BOEHM, 1981). When the software is a programming language implementation, the requirements are described in the language definition and its semantics. This way, the faster we can have feedback about the language basic constructs and their semantics, the better for its development.

Particular implementations — and hardware — can always improve all kinds of non-functional aspects, like performance and security, for instance. But after a language specification is ready, its first implementation is released, and it is widely used, changing the semantics of its core concepts becomes harder as there is software already written based on the original semantics. Next language versions can't break that previously written software.

Prototyping is a technique that is widely used in the context of software development, as a way of discovering early the unavoidable errors in the requirements. Those errors in the requirements can be omissions, inconsistencies, ambiguity or even wrong information. In (RANGEL, 2003), there is a good survey on prototyping and its benefits for the software development activity.

This chapter presents a method for prototyping languages based on their semantic specification, using the Haskell programming language to rapidly build a working prototype of the language. Furthermore, it is shown that denotational semantics can even be expressed directly in Haskell, which is a way of keeping the semantics definition and its prototype implementation synchronized to each other, and bringing some benefits, that are shown in this chapter. Although denotational semantics is the perfect choice for this technique, it is shown that operational semantics specifications can also be used to derive working prototypes, by mapping systematically their rules to Haskell functions.

The remainder of this chapter is organized as follows: Section 5.2 introduces the Haskell programming language; section 5.3 presents a method for deriving prototype implementations from semantics specifications using Haskell; section 5.4 discuss how to make those prototypes closer to what users expect; section 5.5 enumerates related work; section 5.6 ends the chapter discussing results and future work.

5.2 The Haskell programming language and semantic prototyping

Haskell (JONES et al., 2003) is a purely functional programming language. It has static, polymorphic typing and lazy evaluation, among other important features.

What makes Haskell specially suitable for the task of semantics-based prototyping is that its syntax and semantics are very close to those of the lambda calculus. Indeed, Haskell **is** based on the lambda calculus, and can even be translated into lambda calculus, as shown in (DAVIE, 1992).

Writing a Haskell program that implements a denotational semantics (as shown in (WATT, 1991; SCHMIDT, 1986)) is straightforward: it's almost just transliterating the specification into Haskell. In fact, an experienced Haskell programming starting to study denotational semantics, when reading one of those books, would think "hey, this semantics thing is just Haskell programming!". Prototyping operational semantics in Haskell is not as direct as denotational semantics, but it is not difficult.

To illustrate the applicability of Haskell on rapid prototyping of denotational semantics, consider the following denotational definition:

$$
\begin{array}{l}
\mathit{eval}\ [\![e_1; e_2]\!]\ \mathit{env}\ \mathit{sto} = \\
\quad \textbf{let}\ (\_, \mathit{env}', \mathit{sto}') = \mathit{eval}\ e_1\ \mathit{env}\ \mathit{sto} \\
\quad \textbf{in}\ \mathit{eval}\ e_2\ \mathit{env}'\ \mathit{sto}'
\end{array}
$$

The definition can be implemented in Haskell as follows:

```
1 eval (Seq e1 e2) env sto =
2   let (_, env', sto') = eval e1 env sto
3   in eval e2 env' sto'
```

As one can see, implementing a mathematical semantics definition is quite direct. In the case of denotational semantics, it involves mainly minor syntactic transformation.

In the case of operational semantics, as we shall see, things get a little bit more complicated. In this chapter we present a sistematic way of going from operational semantics definition to Haskell code, that works for simpler (i.e. deterministic) operational semantics.

5.3 Developing semantics-based prototype implementations

This section illustrates semantics-based prototyping with Haskell by defining a toy language, in terms of its abstract syntax and semantics, and developing prototype implementations for it in Haskell, using both denotational and operational semantics. Complete source code for presented definitions and implementations is available on the internet (AZEVEDO TERCEIRO, 2006).

At the risk of boring the reader, after each block of definitions, we show the Haskell code that implements them, with the goal of illustrating how that implementation can be derived from the mathematical definitions.

Without any creativity, our toy language will be called *toy*. It is a dynamically-typed imperative language, whose types are *nil*, natural numbers and functions. All these types are first-class values: they can be assigned to variables, passed to and returned from functions.

toy is kept simple to the extreme, so functions are always unary. But since functions are first-class values, multiple parameters can be simulated by using higher-order functions.

As experienced Lua users may note, *toy* is based on Lua (IERUSALIMSCHY, 2003), with several simplifications and omissions.

5.3.1 Abstract syntax

As said before, in *toy* values can be nil (representing no value at all), integer numbers or functions, and all of them are first-class values.

$$Value \rightarrow nil \mid \mathbb{N} \mid \lambda x.e$$

The basic building block in *toy* are expressions. They can be values, variables, function calls, assignments, conditionals, sequencings (an expression followed by another expression), and binary operations.

$$\begin{aligned} Exp \rightarrow\ & Value \mid x \mid e_1 e_2 \mid x = e \\ & \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ end} \mid e_1; e_2 \mid e_1 \oplus e_2 \end{aligned}$$

In the above definition, $\oplus \in Operator$ stands for binary operators. For now, we'll stick to just sum and multiplication as the possible binary operations:

$$Operator \rightarrow + \mid *$$

What we've seen so far gives us *toy*'s abstract syntax. The above definitions fully specify the main elements that can be used to build *toy* programs. As most abstract syntax definitions, it is ambiguous (WATT, 1991). But for now, what's important is the structural relation between the elements of the language: what elements are part of the others, and what other elements one element is composed of.

This abstract syntax definition can be easily coded in Haskell as follows (where `Id` is a type synonym for `String`):

```
1 data Value = Nil | Number Int | Function Id Exp
2 data Exp = Val Value | Var Id | Call Exp Exp | Assign Id Exp
3          | Cond Exp Exp Exp | Seq Exp Exp | Op Operator Exp Exp
4 data Operator = Sum | Mult
```

In line 1 is created a data type for values, defined as a tagged union of the singleton nil value, integer numbers, and functions (`Function`). As stated before, functions are unary. Lines 2–3 define the data type for expressions and line 4 does the same for binary operators.

We highlight here how straightforward this implementation is, given the mathematical definition.

5.3.2 Basic semantic domains

As a first step, we have to define some semantic domains, in terms of which — together with the abstract syntax definition — we'll give semantics to our language.

Our first semantic domain is a model for storage. This domain aims to model memory in a (very) simplified way: $Store$ is a Cartesian product domain of functions from $Location$ to $Value$ ($Location \rightarrow Value$) and $Location$, where $Location$ is just a meaningful name for integer numbers. We use that second component as an indicator of the next available empty memory cell.

$$Store = (Location \rightarrow Value) \times (Location)$$

$$Location = \mathbb{N}$$

S_0, the empty store, has undefined values in all locations.

$$S_0 : Store$$
$$S_0 = (\lambda x.\bot, 0)$$

Now we have to define semantic functions that we'll use to manipulate stores. Our first semantic function is $fetch$: it returns the value stored in a particular location:

$$fetch : Store \to Location \to Value$$

$$fetch\ (f, n)\ l = f\ l$$

$update$ stores a particular value in a particular location:

$$update : Store \to Location \to Value \to Store$$

$$update\ (f, n)\ l\ v = ((\lambda loc.\mathbf{if}\ loc \equiv l\ \mathbf{then}\ v\ \mathbf{else}\ f\ loc), n)$$

Our last function is $alloc$: it allocates a new location to be used, and yields that location together with an modified store:

$$alloc : Store \to Location \times Store$$

$$alloc\ (f, n) = (n, (f, n + 1))$$

Note that the function component of the new store is just equal to that of the old one. The newly allocated cell, although being reserved for use, has no associated value.

The $Store$ domain is, as said before, a very simplified model of memory. It does not feature deallocation, for example. Also, the memory amount needed for storing each type of value is not considered: we just assume they all fit in a "memory cell". It also does not consider any implementation concerns that would show up when writing an actual implementation of a memory management system.

In spite of those simplifications, this definition is functional enough for defining a proper semantic specification for our language, while abstracting low-level details of an actual implementation.

After having a suitable model for memory, we need a model for names and scopes. A common model for scope of variables – and names in general – are *environments*, functions from identifiers to language entities, such as variables, functions, classes, modules, etc.

As in *toy* all values are first-class, we can take environments to be just functions from identifiers to locations, allowing an undefined value as an outcome of such functions (that's the case, for example, when we try to access an inexistent variable x).

$$Environment = Id \to Location_\bot$$

E_0, the empty environment, maps all identifiers to an undefined value.

$$E_0 : Environment$$
$$E_0 = \lambda x.\bot$$

Given an environment e, $e\,[x \mapsto v]$ represents an environment that is exactly equal to e, except that it maps (or binds) x to v. $e\,[x \mapsto v]$ can be defined as:

$$e\,[x \mapsto v] = \lambda y.\mathbf{if}\ x \equiv y\ \mathbf{then}\ v\ \mathbf{else}\ e\ y$$

It's worth to note that in this definition, we are interpreting $\perp$ as an undefined value, and not considering the possibility of nontermination. Thus, we can always test if some value yielded by an Environment function is $\perp$ or not. Unless caught by such a test, a Environment function yielding $\perp$ means a runtime error (what should be expect as the outcome of trying to reference an undefined variable).

The definitions for the presented semantic domains can be implemented in Haskell as follows:

```
1  type Store = Pair (Location -> Value) (Location)
2  type Location = Int
3  emptyStore :: Store
4  emptyStore = (\ x -> error "invalid location", 0)
5  fetch :: Store -> Location -> Value
6  fetch (f,n) l = f l
7  update :: Store -> Location -> Value -> Store
8  update (f,n) l v = ((\ loc -> if loc == l then v else f loc), n)
9  alloc :: Store -> Pair Location Store
10 alloc (f,n) = (n, (f, n+1))
11 type Environment = Id -> Maybe Location
12 emptyEnv :: Environment
13 emptyEnv = \ x -> Nothing
14 bind x v e = \ y -> if x == y then Just v else e y
```

Again we highlight how straightforward this implementation is, based on the mathematical notation.

Lines 1–10 implement the $Store$ domain. As before, the Haskell code is almost the same as the mathematical definition, with just some minor syntax issues. Lines 11–14 implement the $Environment$ domain. The binding operation $e\,[x \mapsto v]$ is represented as `bind x v e` and implemented by the `bind` function.

5.3.3 A denotational semantics for *toy*

Now that we have both an abstract syntax definition and suitable semantic domains, we can specify *toy*'s actual semantics, i.e., give meaning for the language elements. For this we'll use an enriched version of the lambda calculus, with *if* and *case* expressions. As shown in (WATT, 1991; DAVIE, 1992), those constructs can be defined in terms of the core lambda calculus. Thus, despite our somewhat richer syntax, we're still in lambda calculus' realm. [1]

An expression in *toy* has the effect of producing a value, perhaps changing the environment (by adding variables to it) and/or changing the store (by updating or allocating new cells in it). We capture this meaning with the $eval$ function:

$$eval : Exp \rightarrow Environment \rightarrow Store \rightarrow (Value \times Environment \times Store)$$

[1] Although the recursive applications of $eval$ in the definition of $eval$ itself would require some fixedpoint combinators in the real lambda calculus. (WATT, 1991; SCHMIDT, 1986)

Now we have to define $eval$ regarding each one of the alternatives in Exp's definition. We start with $[\![v]\!]$: evaluating a value just yields that value, without changing environment or store:

$$eval\ [\![v]\!]\ env\ sto = (v, env, sto)$$

Evaluating a variable means to check if it was defined previously. If not, this is an error. Otherwise, we fetch from memory the contents stored in the location associated with that variable.

$$\begin{array}{l} eval\ [\![x]\!]\ env\ sto = \\ \quad \textbf{case}\ env\ x\ \textbf{of} \\ \quad\quad \bot \rightarrow \bot \\ \quad\quad loc \rightarrow (v, env, sto)\ \textbf{where}\ v = fetch\ sto\ loc \end{array}$$

A function call (or application) means first evaluating the first expression (which is supposed to yield a function value), then evaluating the second expression in the resulting environment and store, and then using the auxiliary function $apply$ to actually apply v_2 to v_1. The $apply$ function is defined later; for now, we just assume that it will yield the expected value, together with potentially modified environment and store.

$$\begin{array}{l} eval\ [\![e_1 e_2]\!]\ env\ sto = \\ \quad \textbf{let}\ (v_1, env', sto') = eval\ e_1\ env\ sto \\ \quad\quad\ (v_2, env'', sto'') = eval\ e_2\ env''\ sto'' \\ \quad \textbf{in}\ apply\ v_1\ v_2\ env''\ sto'' \end{array}$$

Assignments can have two possible effects: if there is no variable with that name, it binds the variable to a new memory cell and store the computed value there. In the case when the variable already exists, it is just updated. An assignment yields the value computed from the right hand side expression.

$$\begin{array}{l} eval\ [\![x = e]\!]\ env\ sto = \\ \quad \textbf{let}\ (v, env', sto') = eval\ e\ env\ sto \\ \quad \textbf{in case}\ env'\ x\ \textbf{of} \\ \quad\quad \bot \rightarrow \textbf{let}\ (loc, sto'') = alloc\ sto' \\ \quad\quad\quad\quad \textbf{in}\ (v, env'\,[x \mapsto loc]\,, update\ sto''\ loc\ v) \\ \quad\quad loc \rightarrow (v, env', update\ sto'\ loc\ v) \end{array}$$

Conditional expressions are evaluated as follows: first e_1 (the condition) is evaluated; if it yields $[\![nil]\!]$ then e_3 is evaluated, with e_2 being evaluated otherwise.

$$\begin{array}{l} eval\ [\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \textbf{end}]\!]\ env\ sto = \\ \quad \textbf{let}\ (v, env', sto') = eval\ e_1\ env\ sto \\ \quad \textbf{in case}\ v\ \textbf{of} \\ \quad\quad\quad [\![nil]\!] \rightarrow eval\ e_3\ env'\ sto' \\ \quad\quad\quad x \rightarrow eval\ e_2\ env'\ sto' \end{array}$$

Sequencing is trivial: e_1 is evaluated, yielding a value together with potentially modified environment and store. That value is discarded and e_2 is evaluated in the new environment/store context.

$$\begin{array}{l} eval\ [\![e_1; e_2]\!]\ env\ sto = \\ \quad \textbf{let}\ (\_, env', sto') = eval\ e_1\ env\ sto \\ \quad \textbf{in}\ eval\ e_2\ env'\ sto' \end{array}$$

Binary operators are handled by evaluating e_1 and e_2 sequentially [2], and then the actual calculation is done by the *operate* function, which depends on the actual operator being evaluated. *operate* is defined later, for now we just trust it to yield the operation's result.

$$
\begin{array}{l}
eval\ [\![e_1 \oplus e_2]\!]\ env\ sto = \\
\quad \textbf{let}\ (v_1, env', sto') = eval\ e_1\ env\ sto \\
\qquad\quad (v_2, env'', sto'') = eval\ e_2\ env'\ sto' \\
\quad \textbf{in}\ (operate\ \oplus\ v_1\ v_2, env'', sto'')
\end{array}
$$

Now what's left are the previously used auxiliary functions. *apply*, the helper function for evaluating function calls ($[\![\lambda x.e]\!]\ v$) is in the following domain:

$$apply : Value \to Value \to Environment \to Store \to (Value \times Environment \times Store)$$

But among *Value* alternatives, *apply* is only defined for functions. It creates a new environment binding the formal parameter to the actual parameter, and evaluates the function body (e) in the new environment. *apply* yields the value yielded by the evaluation of the function body and the (possibly) modified store. That new environment is discarded, since the formal parameter, as well as everything created inside the function body has local scope.

$$
\begin{array}{l}
apply\ [\![\lambda x.e]\!]\ v\ env\ sto = \\
\quad \textbf{let}\ (loc, sto') = alloc\ sto \\
\qquad\quad (v', \_, sto'') = eval\ e\ (env\ [x \mapsto loc])\ (update\ sto'\ loc\ v) \\
\quad \textbf{in}\ (v', env, sto'')
\end{array}
$$

If, in $[\![e_1 e_2]\!]$, e_1 does not yield a function, the whole expression has an undefined semantics (although being syntactically valid).

The *operate* function used in evaluation of binary operations is as follows:

$$operate : Operator \to Value \to Value \to Value$$

And it is defined for operators $[\![+]\!]$ and $[\![*]\!]$, together with number arguments. *operate* is undefined when x or y are not both numbers.

$$
\begin{array}{l}
operate\ [\![+]\!]\ [\![x]\!]\ [\![y]\!] = [\![(x + y)]\!] \\
operate\ [\![*]\!]\ [\![x]\!]\ [\![y]\!] = [\![(x \times y)]\!]
\end{array}
$$

We have just presented *toy*'s denotational semantics, giving meaning to the language constructs in terms of mathematical domains and functions. This semantics can be implemented in Haskell as follows (where `Triple a b c = (a,b,c)`).

```
1 eval :: Exp -> Environment -> Store -> (Triple Value Environment
     Store)
2 eval (Val v) env sto = (v, env, sto)
3 eval (Var x) env sto =
4   case env x of
5     Nothing -> error ("undefined variable \"" ++ x ++ "\"")
6     Just loc -> (v, env, sto) where v = fetch sto loc
7 eval (Call e1 e2) env sto =
```

[2] since expressions in *toy* have side-effects, we have to specify an evaluation order to simplify our semantics. In an actual implementation, however, the expressions could be evaluated in parallel, depending on some static analysis made to check if, for example, the expressions depend on each other.

```
 8   let (v1, env', sto') = eval e1 env sto
 9       (v2, env'', sto'') = eval e2 env'' sto''
10   in apply v1 v2 env'' sto''
11 eval (Assign x e) env sto =
12   let (v,env',sto') = eval e env sto
13   in case env' x of
14     Nothing -> let (loc,sto'') = alloc sto'
15                in (v, bind x loc env', update sto'' loc v)
16     Just loc -> (v, env', update sto' loc v)
17 eval (Cond e1 e2 e3) env sto =
18   let (v, env', sto') = eval e1 env sto
19   in case v of
20        Nil -> eval e3 env' sto'
21        x   -> eval e2 env' sto'
22 eval (Seq e1 e2) env sto =
23   let (_, env', sto') = eval e1 env sto
24   in eval e2 env' sto'
25 eval (Op op e1 e2) env sto =
26   let (v1,env',sto') = eval e1 env sto
27       (v2,env'',sto'') = eval e2 env' sto'
28   in (operate op v1 v2, env'', sto'')
29 apply :: Value -> Value -> Environment -> Store -> (Triple Value
       Environment Store)
30 apply (Function x e) v env sto =
31   let (loc, sto') = alloc sto
32       (v',_,sto'') = eval e (bind x loc env) (update sto' loc v)
33   in (v',env,sto'')
34 operate :: Operator -> Value -> Value -> Value
35 operate Sum      (Number x) (Number y) = Number (x + y)
36 operate Mult (Number x) (Number y) = Number (x * y)
```

The above Haskell code implements the given semantics by almost "just" transliterating the mathematical definition into Haskell. In more detail, we have:

- line 2: $eval\ [\![v]\!]\ env\ sto$
- lines 3–6: $eval\ [\![x]\!]\ env\ sto$
- lines 7–10: $eval\ [\![e_1 e_2]\!]\ env\ sto$
- lines 11–16: $eval\ [\![x = e]\!]\ env\ sto$
- lines 17–21: $eval\ [\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \textbf{end}]\!]\ env\ sto$
- lines 22–24: $eval\ [\![e_1; e_2]\!]\ env\ sto$
- lines 25–28: $eval\ [\![e_1 \oplus e_2]\!]\ env\ sto$
- lines 30–33: $apply\ [\![\lambda x.e]\!]\ env\ sto$
- line 35: $operate\ [\![+]\!]\ [\![x]\!]\ [\![y]\!]$
- line 36: $operate\ [\![*]\!]\ [\![x]\!]\ [\![y]\!]$

5.3.4 An operational semantics for *toy*

To give an operational semantics (PLOTKIN, 1981) to *toy*, we need first to define a tuple representing configurations:

$$C = Exp \times Environment \times Store$$

The semantics itself is given in terms of a relation $\rightarrow \subseteq C \times C$, where $\gamma \rightarrow \gamma'$ means "there is a transition from configuration γ to configuration γ'". The $\rightarrow$ relation is then defined in terms of rules.

Different from before, in this section we show an operational semantics for *toy* while immediately following groups of rules by its Haskell implementation. For both rules definitions and Haskell code, we'll be reusing our previously defined semantic domains for environments and stores.

To transform the operational semantics rules into Haskell functions, we can use a transformation T, that for each rule in operational semantics, gives us the rule's implementation in Haskell as a `trans` function:

$$\begin{array}{rcl} T\left(\gamma \rightarrow \gamma'\right) & \stackrel{def}{=} & trans\ \gamma = \gamma' \\ T\left(\frac{\beta_1 \rightarrow \beta_1', \ldots, \beta_n \rightarrow \beta_n'}{\gamma \rightarrow \gamma'}\right) & \stackrel{def}{=} & \left[\begin{array}{l} trans\ \gamma = \gamma' \\ \quad where\ \beta_1' = trans\ \beta_1, \ldots, \beta_n' = trans\ \beta_n \end{array}\right. \end{array}$$

This transformation T is useful for prototyping deterministic operational semantics (i.e., semantics where if $\gamma \rightarrow \gamma'$ and $\gamma \rightarrow \gamma''$, then $\gamma' = \gamma''$). This is the case of the *toy* operational semantics presented here.

Non-derministic operational semantics are a little bit more complicated. Section 5.5 points to literature where the issues involved are discussed.

In a case by case basis, we'll show explicitly how T gives us the Haskell implementation of a rule, as well as the variations of T needed in special cases..

We have to start our Haskell implementation by defining the data type for configurations, and declaring the `trans` function.

```
1 data Config = C Exp Environment Store
2 trans :: Config -> Config
```

Now let's begin the operational semantics definition and its implementation: variables are the simplest case.

$$\langle x, en, st\rangle \rightarrow \langle fetch\ st\ (en\ x), en, st\rangle \tag{5.1}$$

The rule in definition 5.1 translates into Haskell using T's first case : [3]

```
1 trans (C (Var x) en st) =
2   C (Val (fetch st (fromJust (en x)))) en st
```

$$\langle (\lambda x.e)\ v, en, st\rangle \rightarrow \langle e, en\left[x \mapsto loc\right], update\ st'\ loc\ v\rangle \tag{5.2}$$

where $(loc, st') = alloc\ st$

[3] We just assume that $x \in domain(en)$. If it's not the case, the `fromJust` application will cause the Haskell program to crash (what is expected from a program that tries to reference an undefi ned variable!)

$$\frac{\langle e_2, en, st\rangle \to \langle e_2', en', st'\rangle}{\langle v\ e_2, en, st\rangle \to \langle v\ e_2', en', st'\rangle} \qquad \frac{\langle e_1, en, st\rangle \to \langle e_1', en', st'\rangle}{\langle e_1 e_2, en, st\rangle \to \langle e_1' e_2, en', st'\rangle} \tag{5.3}$$

Rules in definitions 5.2 and 5.3 define function application. We now have to deal with the more complicated cases, when we start to use the second case for the transformation T when writing the Haskell code, plus some variations required by the circumstances:

```
1 trans (C (Call (Val (Function x e)) (Val v)) en st) =
2   C e (bind x loc en) (update st' loc v)
3     where (loc,st') = alloc st
4 trans (C (Call v@(Val(Function x e)) e2) en st) =
5   C (Call v e2') en' st'
6     where C e2' en' st' = trans (C e2 en st)
7 trans (C (Call (Val v) _) en st) =
8   error ("called non-function " ++ (show v))
9 trans (C (Call e1 e2) en st) = C (Call e1' e2) en' st'
10  where C e1' en' st' = trans (C e1 en st)
```

In this case, we needed an extra definition for when the first expression is not a function (lines 7–8), which was implicit in the transition relation. Since in Haskell pattern matching is tested top-down, if that rule is not there our implementation will loop, because `e1` and `e2` in the last definition (lines 9–10) would match any expressions, including non-function values.

$$\langle x = v, en, st\rangle \to \langle v, en_1, update\ st\ (en\ x)\ v\rangle \quad if\ x \in domain(en) \tag{5.4}$$

$$\langle x = v, en, st\rangle \to \langle v, en_1\left[x \mapsto loc\right], update\ st_2\ loc\ v\rangle \quad if\ x \notin domain(en) \tag{5.5}$$

where $(loc, st_2) = alloc\ st_1$

$$\frac{\langle e, en, st\rangle \to \langle e', en', st'\rangle}{\langle x = e, en, st\rangle \to \langle x = e', en', st'\rangle} \tag{5.6}$$

The rules in definitions 5.4, 5.5 and 5.6 define assignment. Their implementation is slightly different from the conventional, because we chose to represent rules 5.4 and 5.5 in a single definition case of `trans` (lines 1–5), by using a case expression.

```
1 trans (C (Assign x (Val v)) en st) =
2   case en x of
3     Nothing  -> (C (Val v) (bind x loc en) (update st' loc v))
4                   where (loc,st') = alloc st
5     Just loc -> (C (Val v) en (update st loc v))
6 trans (C (Assign x e) en st) = C (Assign x e') en' st'
7   where C e' en' st' = trans (C e en st)
```

$$\langle \textbf{if}\ nil\ \textbf{then}\ e_2\ \textbf{else}\ e_3, en, st\rangle \to \langle e_3, en, st\rangle \tag{5.7}$$

$$\langle \textbf{if}\ v\ \textbf{then}\ e_2\ \textbf{else}\ e_3, en, st\rangle \to \langle e_2, en, st\rangle \tag{5.8}$$

$$\frac{\langle e_1, en, st\rangle \to \langle e_1', en', st'\rangle}{\langle \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, en, st\rangle \to \langle \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3, en', st'\rangle} \tag{5.9}$$

Conditional expressions rules, presented in definitions 5.7, 5.8 and 5.9, are implemented directly through T:

```
1 trans (C (Cond (Val Nil) e2 e3) en st) = C e3 en st
2 trans (C (Cond (Val v) e2 e3) en st) = C e2 en st
3 trans (C (Cond e1 e2 e3) en st) = C (Cond e1' e2 e3) en' st'
4   where C e1' en' st' = trans (C e1 en st)
```

$$\langle v; e, en, st\rangle \to \langle e, en, st\rangle \qquad \frac{\langle e_1, en, st\rangle \to \langle e_1', en', st'\rangle,}{\langle e_1; e_2, en, st\rangle \to \langle e_1'; e_2, en', st'\rangle} \tag{5.10}$$

Rules in definition 5.10 define sequencing expressions, and are also implemented without variation in the usual transformation:

```
1 trans (C (Seq (Val v) e2) en st) = C e2 en st
2 trans (C (Seq e1 e2) en st) = (C (Seq e1' e2) en' st')
3   where C e1' en' st' = trans (C e1 en st)
```

$$\langle v_1 \texttt{+} v_2, en, st\rangle \to \langle v_1 + v_1, en, st\rangle \qquad \langle v_1 \texttt{*} v_2, en, st\rangle \to \langle v_1 \times v_2, en, st\rangle \tag{5.11}$$

where $v_1, v_2 \in \mathbb{N}$

$$\frac{\langle e_2, en, st\rangle \to \langle e_2', en', st'\rangle,}{\langle v_1 \oplus e_2, en, st\rangle \to \langle v_1 \oplus e_2', en', st'\rangle} \qquad \frac{\langle e_1, en, st\rangle \to \langle e_1', en', st'\rangle,}{\langle e_1 \oplus e_2, en, st\rangle \to \langle e_1' \oplus e_2, en', st'\rangle} \tag{5.12}$$

Binary operations, defined by the rules in definitions 5.11 and 5.12, are also implemented without much creativity. The exception is, as in function applications, an extra definition (lines 5–6) to force matching with non-number values and avoid looping in the last rule, that matches against any pair of expressions.

```
1 trans (C (Op Sum (Val (Number v1)) (Val (Number v2))) en st) =
2   C (Val (Number (v1 + v2))) en st
3 trans (C (Op Mult (Val (Number v1)) (Val (Number v2))) en st) =
4   C (Val (Number (v1 * v2))) en st
5 trans (C (Op op (Val v1) (Val v2)) en st) =
6   error "you can only operate numbers!"
7 trans (C (Op op (Val v) e2) en st) =
8   (C (Op op (Val v) e2') en' st')
9     where C e2' en' st' = trans (C e2 en st)
10 trans (C (Op op e1 e2) en st) =
11   (C (Op op e1' e2) en' st')
12     where C e1' en' st' = trans (C e1 en st)
```

To finish the implementation of our operational semantics, we only need to define how to go from an initial configuration to a final one, i.e., how to derivate the whole execution of a *toy* program from individual transitions between configurations. We do

this defining a `derivation` function, that will iterate on the transition relation starting from the initial configuration, producing a list of configurations by applying the transition relation repeatedly: $derivation\ \gamma_0 = \langle \gamma_0, \gamma_1, \ldots, \gamma_n \rangle$, where $\forall i < n, \gamma_i \rightarrow \gamma_{i+1}$, and γ_n is a terminal configuration, i.e., a configuration to which no rule can be applied. It's trivial to realize that terminal configurations are in the form $\langle v, em, st \rangle$, where $v \in Value$.

```
1 terminal :: Config -> Bool
2 terminal (C (Val v) en st) = True
3 terminal c = False
4
5 derivation :: Config -> [Config]
6 derivation conf =
7   if terminal conf
8   then [conf]
9   else conf:(derivation (trans conf))
```

If we apply `derivation` to an initial configuration $\langle e, em, st \rangle$, it will yield a list comprising all the intermediate configurations of a system during the execution of program e.

5.4 More elaborated prototypes

We have presented both a denotational and an operational semantics for *toy*, together with their Haskell implementation. With this in hands, we can already try *toy* out. We just need to fire a Haskell interpreter and experiment selected expressions, environments and stores:

```
$ ghci Denotational.lhs
> let (v,e,s) = eval (Op Sum (Var "x") (Var "x")) (bind "x"0 emptyEnv) (update
emptyStore 0 (NumberLiteral 5)) in v
10
$ ghci Operational.lhs
> derivation (C (Op Sum (Var "x") (Val (Number 1))) (bind "x"0 emptyEnv) (update
emptyStore 0 (Number 6)))
[<x + 1,(en),(st)>,<6 + 1,(en),(st)>,<7,(en),(st)>]
```

Although we have already full implementations of our language, that's not enough to have feedback from its potential users. Representing abstract syntax trees of the language as Haskell expressions is far from being productive. While developing this study, Happy parsers (GILL; MARLOW, 2006) were used to implement concrete syntaxes, and they showed to be very straightforward to write.

5.5 Related work

The general ideas on this chapter are not new. In the paper "Definitional Interpreters for Higher-Order Programming Languages", Reynolds (REYNOLDS, 1972) discussed various issues regarding writing interpreters for higher-order programming languages using functional programming languages that are themselves based on the lambda calculus (i.e., that are also higher-order themselves).

Hartel (HARTEL, 1997) presents a tool called `Latos` for prototyping operational semantics through translation into Miranda functional programs. `Latos` is compared

with other previous tools, such as *ASF+SDF* and *RML* (both referenced in the paper). The presented approach for prototyping operational semantics is much more comprehensive than the presented here. Several aspects and properties of the operational definitions and the restrictions they impose for the prototyping tools are considered. In particular, handling non-deterministic operational semantics is discussed.

Pugs (TANG et al., 2006) is a Perl 6 (WALL et al., 2006) implementation written in Haskell. It's a very young project, but seems to be already the more complete Perl 6 implementation. Pugs has a variety of implemented backends, and apparently will be used to bootstrap Perl 6. This work show the applicability of Haskell on implementation of other languages.

The general idea on how to implement operational semantics in Haskell was borrowed from Holyer, Gallagher and Muller (HOLYER; GALLAGHER; MULLER, 2006), although they don't present a systematic manner to go from the transition relation in mathematical notation to a Haskell implementation.

Moura and colleagues present in (MOURA; RODRIGUEZ; IERUSALIMSCHY, 2004) a concise operational semantics for coroutines, which can be combined with the techniques presented here to achieve a richer language, even in the prototyping stage.

Leal and colleagues present in (LEAL; IERUSALIMSCHY, 2005) an operational semantics for garbage collection and finalizers, which can be used together with our techniques — and a different model of memory — to produce more efficient prototypes.

5.6 Final remarks

Haskell is a good choice for rapid prototyping of languages. This work raised some issues from the experience of implementing working prototypes based on both denotational and operational semantics of languages, so we could achieve some conclusions and compare the use of Haskell on prototyping both types of specification.

Since Haskell supports a limited (but useful) form of literate programming (KNUTH, 1984), using a tool like lhs2TeX (LöH, 2005) — a Haskell/TEX translator with extensive formatting support — allows us to write at once both definition and implementation of denotational semantics specifications. Being able to write semantics directly in Haskell while not loosing the mathematical aspect bring two main benefits: (i) definition and implementation are always synchronized to each other ; (ii) the definition can be type-checked by any Haskell compiler, giving us several tools for writing better and more correct semantic specifications. For instance, the definitions in this chapter's section on denotational semantics are actually written in Haskell and converted to a nicer, more math-like, notation by lhs2TeX; that section, the abstract syntax section and the semantic domains one are, at the same time, text and program. The definitions, however, are still in lambda calculus, and can be manipulated in proofs as always.

Expressing operational semantics in Haskell is not as straightforward as it is with denotational semantics, but as operational semantics tend to be concise, the syntactic gap does little harm (against no harm at all in the case of denotational semantics). But by its nature, operational semantics allows us to abstract certain details that denotational semantics doesn't. For example, in this chapter one can see that the operational definition is cleaner than the denotational one, although both can be successfully implemented. The representation issue deserves some more investigation. Perhaps using Monads (WADLER, 1992) one can find a way of implementing operational definitions in Haskell that is more close syntactically to the mathematical definition. This would allow

us to write operational semantics as literate Haskell programs, what couldn't be done in this chapter.

This work also showed us that several aspects of languages prototyping can be united into a common framework: input files handling, interactive session console, transition systems implementation (as shown in section 5.3.3). We have already started to work on extracting a Semantic Prototyping Framework from our experiments (AZEVEDO TERCEIRO, 2006), aiming to remove the implementor's need of dealing with those non-functional aspects of the prototype.

The next chapter presents a semantics-based prototype for Algebraic Prosoft that was developed following the methodology presented in this chapter.

6 THE DEVELOPMENT OF A SEMANTICS-BASED PROTOTYPE FOR ALGEBRAIC PROSOFT

In this work, it was developed a semantics-based prototype for Algebraic Prosoft, using the techniques presented in chapter 5.

This chapter describes some concepts that are crucial in the development of full semantics-based prototype implementations, together with some implementation aspects of Algebraic Prosoft's prototype.

Section 6.1 presents the concept of Literate Programming. Section 6.2 presents some aspects on the use of the Haskell Programming language together with Literate Programming techniques for semantic prototyping. Section 6.3 discusses the structure of the prototype implementation of Algebraic Prosoft. Section 6.5 summarizes this chapter.

6.1 Literate Programming

Literate Programming is a technique for typesetting programs in a way they are best presented for people. It reverts the regular logic behind most of programming habits: instead of writing a compiler-centric program, one writes a formatted document as it were mainly intended to be read by people.

Literate Programming was introduced by Donald E. Knuth, in his seminal paper "Literate programming"(KNUTH, 1984). Knuth writes:

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Knuth's intent with the development of the WEB system, described in that paper, was to make the work of writing programs a combination of formal and informal language, in order to allow one to write program in the best way for human understanding, abstracting syntactic restrictions.

The main idea is that literate programs are composed of chunks, and each chunk can be either a textual documentation, or actual source code. And the same literate program can be given as input to both a compiler and a document preparation system, maybe after some automatic preprocessing.

According to Norman Ramsey in (THOMPSON, 2000), the main characteristics of Literate Programming are:

- **Flexible order of elaboration.** The program can be written in any order, preferably that one that makes reading by people easier.

- **Typeset documentation, especially diagrams and mathematics.** In many cases, being able to typeset diagrams and mathematics is very useful to precisely describe complex programs.

- **Automatic support for browsing.** Having a table of contents, indexes, cross-references and other navigation elements is crucial for a book-quality document.

The last two characteristics are accomplished by using a good document preparation system. In fact, most of the literate programming systems do use some TEX variant, which gives those literate programming system all richness that TEX already brings to regular typesetting.

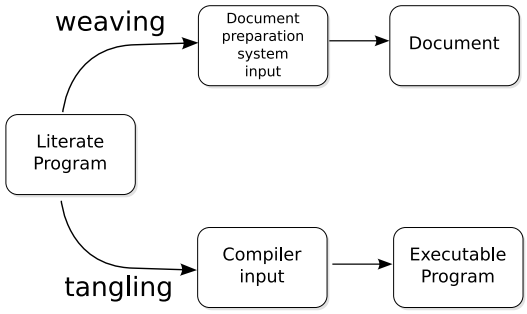


Figure 6.1: tool support for literate programming

For actual literate programming systems, tool support is an important issue. (THOMPSON, 2000) cites several available literate programming tools. Figure 6.1 shows how one achieves documents and programs from a literate program: there is a tool that generates input for a document preparation system from the literate program; this is called **weaving**. [1] Another tool generates input for the compiler from the literate program; this is called **tangling**.

In some literate programming systems those intermediate steps may not exist: either the compiler also does the tangling, or there is no need for preprocessing of the literate programming before feeding it to the document preparation system.

6.2 Literate Programming and the Haskell Programming Language

Haskell's syntax is very similar to lambda calculus, and in particular to the style used commonly (SCHMIDT, 1986; WATT, 1991) to describe Denotational Semantics. This makes Haskell a good tool for prototyping denotational semantics definitions: one can write a denotational semantics as a Haskell program almost with no syntactic changes; and by adding an input parser and some minor code for user interaction, one can obtain a running executable for that semantics.

Haskell also supports a limited form of Literate Programming: literate programs are officially defined in Haskell definition (JONES et al., 2003), and can be written by naming the source file according to the adopted convention for literate Haskell source (`*.lhs`) and starting every source code line with a "`>`" sign: all other lines will be considered as comments by compilers. This way, *tangling* is natively supported by Haskell compilers.

[1] Recently, the term *weaving* has been also used in the area of Aspect-Oriented Software Development.

There are several *weaving* tools for Haskell, but for this work `lhs2TeX` (LöH, 2005) was chosen. It supports the use of TEX for formatting the documentation sections of the program, and generates special TEX code for source-code sections, so Haskell source can be made stylized as regular mathematics in the generated document.

`lhs2TeX` provides also the possibility of detailed formatting of several aspects of the code, with the intent of bringing the semantics expressed in Haskell for practical reasons yet closer to mathematics. For example, we can instruct `lhs2TeX` to present the Haskell expression `Data.Set.empty` as the empty set symbol $\emptyset$, or `Data.Set.union a b` as $a \cup b$.

`lhs2TeX` generates a TEX document from an literate Haskell program, playing the *weaving* role (see figure 6.1).

This way, Haskell seemed a natural choice of tool for prototyping Algebraic Prosoft Semantics. Actually, Algebraic Prosoft's semantics was written directly in Haskell, so both the text *and* the prototype, at the end, are the same thing. This showed some advantages:

- **Easier typesetting.** Writing Haskell is easier than writing lambda calculus in TEX with the proper indentation and formatting.
- **Easier maintainance.** Semantics definition and prototype implementation are always synchronized.
- **Semantics is kept consistent.** The documented semantics can be checked by the Haskell compiler, and won't show any syntax errors, type errors or similar errors.

6.3 The prototype

The prototype implementation of Algebraic Prosoft was developed using the techniques presented in chapter 5, together with literate programming in Haskell for writing the semantics itself. As discussed in chapter 5, it cannot be considered a production implementation, not addressing issues as performance, security and all other non-functional — but important — aspects. On the other hand, it's indeed a full implementation of Algebraic Prosoft, since the language can be actually used and tested through that implementation.

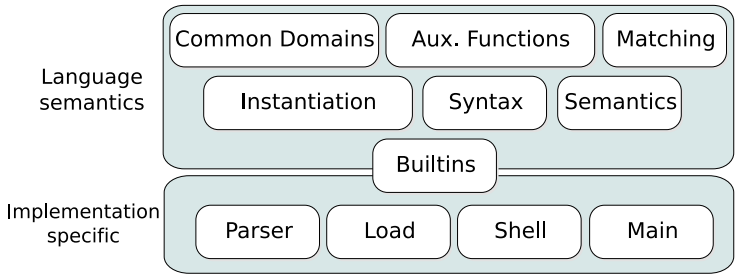


Figure 6.2: `prosoft-reduce`'s modules

The prototype was called `prosoft-reduce`. Figure 6.3 shows in a simplified way `prosoft-reduce`'s modules and their role. The top of the figure comprises the modules that are also the semantics definition: each one of them is described in chapter 4. In the bottom of the figure, there are the modules that are implementation-specific:

- `Parser`: this module is a Happy parser (GILL; MARLOW, 2006) for Algebraic Prosoft specifications and terms. Additionally, it contains also a parser for console commands made available by the `Main` module.

- `Main`: this module is the main mechanism that drives the execution of the prototype. It implements the main loop, reading input from the user, issuing the input to the semantics modules, and presenting results from the users. It also implements the special commands, like `/load`, that loads a new specification into the environment, `/in`, that makes the user enter the scope of a given ATO, and others (see appendix A).

- `Shell`: this module handles user interaction through Haskell's GNU Readline binding, allowing the user to navigate through previously issued terms and commands, and to edit his/her input in a suitable console interface.

- `Load`: handles the loading of ATO's from files, allowing the user to load his/her own specifications into the environment.

The `Builtins` module is in the half of the way: since Records and Unions specifications vary on the number of fields or choices, respectively, we need to dynamically generate specifications based on the number of fields or choices. This way the $include$ semantic function (page 33) needs to use the $createRecordSpec$ and $createUnionSpec$ auxiliary functions, implemented in the `Builtins` module and omitted in the semantics definition. The `Builtins` module also lists and creates the built-in specifications (see appendix B): since they are implemented in Algebraic Prosoft itself, they are not part of the semantics.

Figure 6.3 shows `prosoft-reduce` running a sample session. First, when starting up, `prosoft-reduce` loads all its built-in ATO's. Then the user can reduce terms through the ICS, as shown in the figure. The user can also "jump" to the context of a specific ATO (e.g. `/in Booleans`) and then issue terms that will be reduced in the context of that ATO, just as if they were sent there through ICS.

`prosoft-reduce`'s architecture is shown in figure 6.4: it's a 3-tier application. The layer at the bottom, *Semantics*, was derived from the denotational specification of Algebraic Prosoft's semantics: it contains the modules that form the semantics of the language, as well as its abstract syntax. The middle layer, the *Parser*, interprets the input text using a concrete syntax specification written in Happy (GILL; MARLOW, 2006). This interpretation transforms the input into abstract syntax elements that are fed into the Semantics layer for interpretation. The topmost layer, *Console-based UI*[2], handles the interaction with the user, reading input and presenting results as output to the user.

Since we have clearly separated layers, we can add different implementations of each layer without having much impact on the other ones. Something that is already being worked on is a graphical interface supporting not only graphical editing of Prosoft specifications (like ProTool (RANGEL; NUNES, 2004)), but also graphical representation of terms. As figure 6.4 shows, this graphical interface will be placed as another option of user interface. Its introduction will make specification and specification prototyping even more productive.

The appendices of this thesis present further information on `prosoft-reduce`. Appendix A presents a simple reference manual for `prosoft-reduce`. Appendix B

[2]User Interface

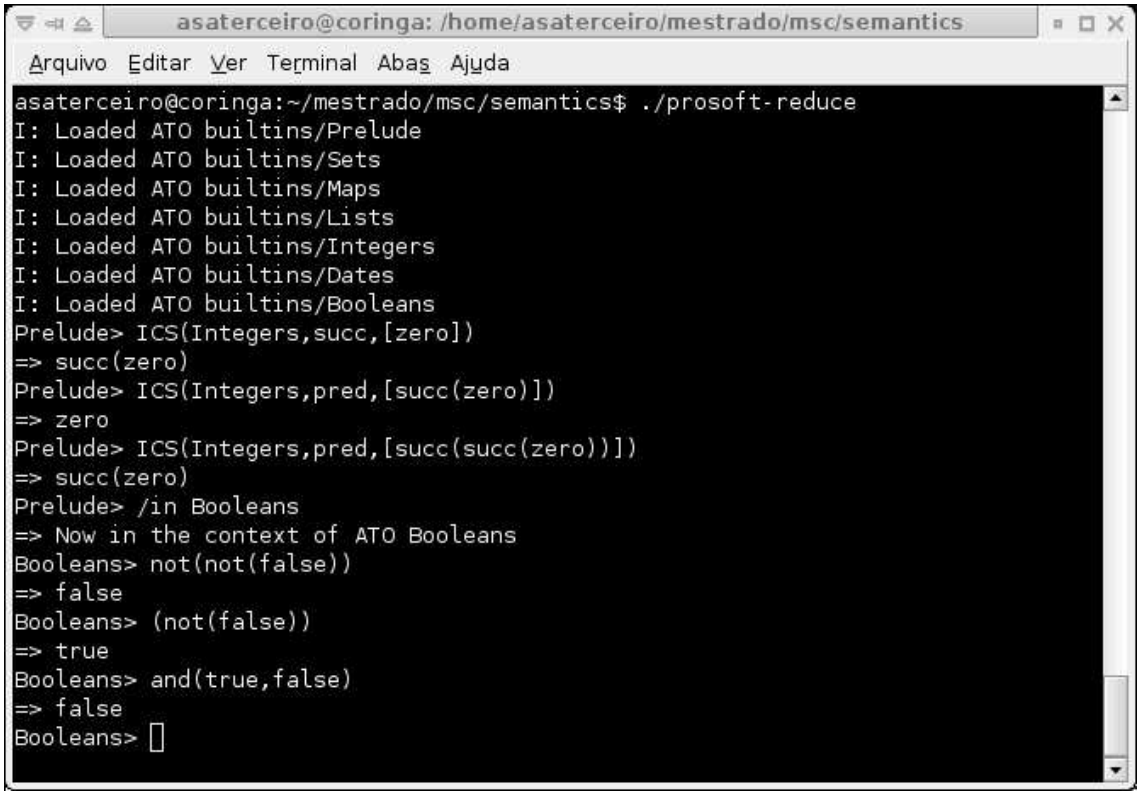


Figure 6.3: A screenshot of `prosoft-reduce`

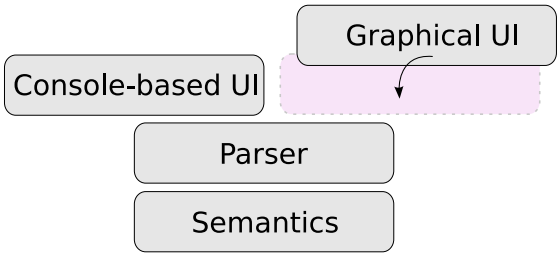


Figure 6.4: `prosoft-reduce`'s architecture

exhibits the implementation of the standard (or built-in) ATO for `prosoft-reduce`, written in Algebraic Prosoft itself. Appendix C revisits the VideoClub problem, a historical example used for illustrating Algebraic Prosoft, this time from the point of view of the "new" interpretation for Algebraic Prosoft.

6.4 A Semantic Prototype Framework

During the development of `prosoft-reduce` and the research on semantic prototyping in general, it was noted that several aspects were always present in semantic prototypes. Some of them were extracted into a framework that was called SPF (Semantic Prototyping Framework).

SPF's main features are:

- A driver function, that implements all the generic aspects of a semantic prototype:

if there is no input file, it fires a interactive console session and incrementally reads inputs, parse them, and feed the parsed structures into the semantics module; if there is an input file, the file is fully parsed and then fed into the semantics module. The prototype implementor passes to the driver function both the parsing function and the semantics function.

- Interactive console session: history handling, input line editing, and virtually all features of GNU Readline.
- File input handling: the prototype implementor does not have to worry about dealing with input file. SPF already reads the input file (if there is one), and feeds the parsed program into the semantics module.
- A simple implementation of a transition system for Operational Semantics prototypes: the prototype implementor just has to define his/her configuration data type, and implement its functions for transition (the transition rules) and for checking termination (so the transition system knows when to stop: when it finds a configuration that is considered a terminal one).

To develop a semantic prototype with SPF, the prototype implementor needs to develop, besides the abstract syntax and semantics definitions, a parser function $parser : String \rightarrow Expression$, where $Expression$ is the type of the parsed elements: depending on the language, it can represent commands, expressions, etc. In the case of `prosoft-reduce`, for example, an $Expression$ is either a term to be reduced or a console command to be executed by the program, like "load a new ATO", "test matching terms x and y", etc.

More information on SPF can be found on its internet page (AZEVEDO TERCEIRO, 2006).

6.5 Final remarks

This chapter presented briefly the main aspects involved in the development of a semantics-based prototype implementation of Algebraic Prosoft. It presented the concept of Literate Programming and a discipline of using Literate Programming with the Haskell programming language.

An overview of the prototype was given, describing superficially its main implementation-specific modules, which together with the semantics modules, form `prosoft-reduce`. It was also shown that several aspects of semantic prototypes are common between several implementations, and that they are being extracted into the Semantic Prototype Framework, an ongoing project.

The next chapter finishes this thesis, presenting related work, pointing its main contributions, its limitations, and future work.

7 CONCLUSIONS

This chapter presents the conclusions of this work: what other work is related to this one; how this work contributed to research in its field, as well as in the Prosoft research group; what limitations we choose to live with; and what can be done in the future as a consequence of this work.

The rest of the chapter is organized as follows: section 7.1 points related work; section 7.2 indicates the contributions presented in this work; section 7.3 describes the limitations to which this work is bound. secion 7.4 presents possibilities of future work that can be carried as logical sequences of this one.

7.1 Related work

(RIBEIRO, 1991) presents a formalization of Prosoft in VDM in order to integrate algebraic specifications into the Prosoft Environment. Although modelling a currently outdated version of Prosoft, this work provided some ideas about modelling the Prosoft environment.

ProTool (RANGEL, 2003) is a tool that, through the translation of Prosoft specifications into OBJ (GOGUEN; TARDO, 1986) ones, provides means of prototyping Prosoft specifications. Although being very useful in the process of software development, such prototyping doesn't provide a direct semantics to Prosoft. This way, the manipulation and reasoning over Prosoft specifications wasn't facilitated. It could only be achieved by having a semantics explicitly defined.

7.2 Contributions

This work presents as contributions:

- **Semantics for Algebraic Prosoft.** By defining a precise semantics for Algebraic Prosoft, this work contributes to the Prosoft research group by providing an uniform view of Algebraic Prosoft, unifying its interpretation and providing a reference for it.

  Besides that, the already specified semantics can serve as a base for further development of the Algebraic Prosoft language. Additions to the language can be specified by extending the semantics presented in this thesis.

- **Semantics for ICS.** This contribution can be seen as a more general one, benefitting not only the Prosoft group but the whole formal methods community. ICS is

an unique (to the best of our knowledge) concept that eliminates the need of inclusion of data types by others, allowing a data type to use other data types' operations by referencing (calling) them with a special notation without including the whole corresponding specification. This contrasts with the concept of data type inclusion, when a data type has to include another one as a whole if it uses some of the other's operations. This work presents a precise and unambiguous semantics for the ICS concept.

- **A prototype implementation of Algebraic Prosoft.** Besides specifying semantics, this work also provides a prototype implementation, in which Algebraic Prosoft can be actually experimented and used.

- **Investigation on Semantic prototyping.** This work also provided interesting results investigating the use of the Haskell programming language in semantic prototyping, analyzing the prototyping of both denotational and operational semantics definitions, as shown in chapter 5.

7.3 Limitations

The semantics in chapter 4 is presented in a non-standard way. It mixes semantic and syntactic elements. It does not state explicitly the meaning of Prosoft specifications in terms of a proper mathematical entity. Instead, semantic functions and auxiliary functions aren't properly separated.

This work started with the restriction of using only the prefixed syntax $op(t_1, \ldots, t_n)$ for terms. This syntax form is certainly enough for semantic definition, but makes writing large specifications harder. Allowing mix-fix syntax and other forms of syntactic sugar, like providing a literal representation for some built-in types as Integers and Dates (i.e. `4` instead of `succ(succ(succ(succ(zero))))`) would make Algebraic Prosoft more suitable for a production environment.

It was not specified a static semantics for Algebraic Prosoft. In a production environment it would be crucial to be able to check a specification for type errors. In special, if one tries to reduce a term that does not match any of the operations defined in some ATO, it's simply not reduced. The user could receive a type error message instead, informing that the term is not an object of a sort specified in that ATO.

7.4 Future work

An interesting possibility for future work is further research on semantic prototyping, improving the Semantic Prototyping Framework by identifying other common aspects between prototype implementations and working on a better way of representing operational semantics rules in Haskell.

Since now a precise interpretation of Algebraic Prosoft is available, an obvious future work is to adequate the current implementation of the Prosoft Environment (Prosoft Java) to this interpretation. Prosoft Java relies on an interpretation of Algebraic Prosoft that the Prosoft group currently considers as "not the right one".

Another possibility regarding the Prosoft Environment is to investigate how far can we go with semantic prototyping: would it be possible to integrate `prosoft-reduce` with a graphical interface and other elements of modern user interface design? Can we

integrate the semantics module together with parts of `prosoft-reduce` with existing Prosoft Java user interface?

Finally, having a precise semantics creates the possibility of developing proof of properties over ATO's in the Prosoft Environment. An interesting work would be to attach a theorem prover to the Prosoft Environment.

REFERENCES

AZEVEDO TERCEIRO, A. S. de. **SPF**: a semantic prototyping framework. Available at: <http://www.inf.ufrgs.br/~asaterceiro/spf/>. Visited on April 7th 2006.

BOEHM, B. **Software Engineering Economics**. Englewood Cliffs: Prentice Hall, 1981.

DAHMER, A. **Um Modelo de Processo de Curso**. 2006. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

DAVIE, A. J. **An Introduction to Functional Programming Systems Using Haskell**. [S.l.]: Cambridge University Press, 1992. (Cambridge Computer Science Texts, v.27).

DIJKSTRA, E. W. **A Discipline of Programming**. [S.l.]: Prentice-Hall, 1976.

FREITAS, A. V. P. **APSEE-Global**: um modelo de gerência de processos distribuídos de software. 2005. 255p. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

GILL, A.; MARLOW, S. **Happy**: the parser generator for haskell. Available at: <http://www.haskell.org/happy/>. Visited on January 26th 2006.

GOGUEN, J. V.; TARDO, J. J. An Introduction to OBJ: a language for write and testing formal algebraic program specifications. In: GEHANI, N.; MCGETTRICK, A. D. (Ed.). **Software Specifications Techniques**. [S.l.]: Addison-Wesley Publishing Company, 1986.

GRANVILLE, L. Z.; GASPARY, L. P. **Distributed Prosoft**: management of tools and memory. [S.l.: s.n.], 1996.

GRIES, D. **The Science of Programming**. New York: Springer-Verlag, 1981.

HARTEL, P. **LATOS – a lightweight animation tool for operational semantics**. 1997.

HOLYER, I.; GALLAGHER, J.; MULLER, H. **Lecture Notes for COMS30122 — Advanced Language Engineering**. Available at: <http://www.cs.bris.ac.uk/Teaching/Resources/COMS30122/lectures/>. Visited on January 26th 2006.

IERUSALIMSCHY, R. **Programming in Lua**. [S.l.]: Lua.org, 2003. 288p.

JONES, S. P.; AUGUSTSSON, L.; BARTON, D.; BOUTEL, B.; BURTON, W.; FASEL, J.; HAMMOND, K.; HINZE, R.; HUDAK, P.; HUGHES, J.; JOHNSSON, T.; JONES, M.; ERIK MEIJER, J. L. andy; PETERSON, J.; REID, A.; RUNCIMAN, C.; WADLER, P. **Haskell 98 Language and Libraries**: the revised report. Available at: <http://www.haskell.org/onlinereport/>. Visited on January 9th 2006.

KNUTH, D. E. Literate programming. **The Computer Journal**, Oxford, UK, UK, v.27, n.2, p.97–111, 1984.

LEAL, M. A.; IERUSALIMSCHY, R. A Formal Semantics for Finalizers. **Journal of Universal Computer Science**, [S.l.], v.11, n.7, p.1198–1214, 2005.

LöH, A. **lhs2TeX**. Available at: <http://www.cs.uu.nl/~andres/lhs2tex/>. Visited on October 17th 2005.

MAIA, A. B. **APSEE-Tail**: um modelo de apoio a adaptacao de processos de software. 2005. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

MOURA, A. L. de; RODRIGUEZ, N.; IERUSALIMSCHY, R. Coroutines in Lua. **Journal of Universal Computer Science**, [S.l.], v.10, n.7, p.910–925, 2004.

NUNES, D. J. Estratégia data-driven no desenvolvimento de software. In: SIMPóSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 1992, Gramado. **Anais...** Porto Alegre: Instituto de Informática: UFRGS, 1992. p.81–95.

NUNES, D. J. **PROSOFT**: um ambiente de desenvolvimento de software baseado no método algébrico. [S.l.]: Instituto de Informática – Universidade Federal do Rio Grande do Sul, 1994.

NUNES, D. J. **Projeto PROSOFT – Position Paper**. [S.l.: s.n.], 2003. *Work in progress*.

PLOTKIN, G. D. **A Structural Approach to Operational Semantics**. [S.l.]: Computer Science Department, Aarhus University, 1981.

RANGEL, G. S. **ProTool**: uma ferramenta de prototipação de software para o ambiente PROSOFT. 2003. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática – Universidade Federal do Rio Grande do Sul, Porto Alegre.

RANGEL, G. S.; NUNES, D. J. ProTool: uma ferramenta de prototipação de software para o ambiente prosoft. In: SIMPóSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE – SESSãO DE FERRAMENTAS, 2004. **Anais...** [S.l.: s.n.], 2004.

REIS, C. A. L. **Um gerenciador de processos de software para o ambiente PROSOFT**. 1998. 197p. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REIS, C. A. L. **Uma abordagem flexível para execução de processos de software evolutivos**. 2003. 267p. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REIS, R. Q. **Uma proposta de suporte ao desenvolvimento cooperativo de software no ambiente PROSOFT**. 1998. 177p. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REIS, R. Q. **APSEE-Reuse**: um meta-modelo para apoiar a reutilização de processos de software. 2002. 215p. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REYNOLDS, J. C. Definitional interpreters for higher-order programming languages. In: ACM ANNUAL CONFERENCE, 1972, Boston. **Proceedings...** New York: ACM Press, 1972. p.717–740.

RIBEIRO, L. **Integração no Prosoft de ambientes corretos obtidos a partir de especificações algébricas e executados usando sistemas de reescrita**. 1991. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS.

SANNELLA, D.; TARLECKI, A. Algebraic methods for specification and formal development of programs. **ACM Comput. Surv.**, New York, NY, USA, v.31, n.3, p.10, 1999.

SCHLEBBE, H. **Distributed Prosoft**. [S.l.: s.n.], 1994.

SCHLEBBE, H.; SCHIMPF, S. **Reengineering of PROSOFT in Java**: outubro de 1997 a dezembro de 1997. Porto Alegre: [s.n.], 1997.

SCHMIDT, D. A. **Denotational semantics**: a methodology for language development. Dubuque, IA, USA: William C. Brown Publishers, 1986.

SOUSA, A. L. R. de. **APSEE-Monitor**: um mecanismo de apoio à visualização de processos de software. 2003. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

TANG, A. et al. **Pugs**. Available at: <http://www.pugscode.org/>. Visited on January 26th 2006.

THOMPSON, D. B. **The Literate Programming FAQ**. Available at: <http://www.literateprogramming.com/farticles.html>. Visited on October 14th 2005.

WADLER, P. Monads for functional programming. In: MARKTOBERDORF SUMMER SCHOOL ON PROGRAM DESIGN CALCULI – COMPUTER AND SYSTEMS SCIENCES, 1992. **Proceedings...** [S.l.]: Springer Verlag, 1992. v.18.

WALL, L. et al. **Perl 6**. Available at: <http://dev.perl.org/perl6/>. Visited on January 26th 2006.

WATT, D. A. **Syntax and Semantics of Programming Languages**. [S.l.]: Prentice Hall International, 1991.

APPENDIX A PROSOFT-REDUCE SIMPLIFIED REFERENCE MANUAL

Download and installation

The instructions presented here are mainly targeted at Unix-like environments. Although they were only tested in Debian GNU/Linux systems, they may work on most Unix-like environments with the needed requirements installed, specially with any GNU/Linux variant.

Obtaining `prosoft-reduce`

`prosoft-reduce` can be downloaded from the Prosoft research group's website, under the "Downloads" section. The available file is named `prosoft-reduce-x.y.z.tar.gz`, where `x.y.z` is the version number.

> **Note:**
> To ease the evolution from the viewpoint of users, `prosoft-reduce`'s versions are numbered with numbers that contain three components, in the form $x.y.z$, where:
>
> - x is called *major version number*. It's incremented when there are large, conceptual changes in the design of the software.
> - y is called *minor version number*. It's incremented every time a new version contains substancial, functional changes when comparing to its predecessor.
> - z is called *patch level*. Every version that corrects bugs and non-functional requirements gets a new *patch level* number.

Compilation and installation

Compiling `prosoft-reduce` has the following requirements:

- GNU make.
- Glasgow Haskell Compiler (GHC). `prosoft-reduce` was tested only with versions above 6.4, but there are no special reasons that would make it not work with earlier os later versions. Other Haskell compilers may also work, but were not tested.

The first step is to extract the source code from the tarball file, and enter the source code directory:

```
# tar xvf prosoft-reduce-x.y.z.tar.gz
# cd prosoft-reduce-x.y.z
```

Inside the `prosoft-reduce-x.y.z` directory, you can use the common method for software that is packaged with GNU autotools (most free software out there that are written in a compiled language are):

```
# ./configure
# make
# make install
```

If you want to install `prosoft-reduce` to a privileged path (like `/usr`), the last command must be executed as the `root` user. If you want to install to a user-defined location, you can pass the `--prefix=/your/preferred/path` argument to `./configure`. Try `./configure --help` for a full list of available options.

Usage

If everything went well with the installation procedure, the `prosoft-reduce` command will be immediately available:

```
# prosoft-reduce
I: Loaded ATO builtins/Prelude
I: Loaded ATO builtins/Sets
I: Loaded ATO builtins/Maps
I: Loaded ATO builtins/Lists
I: Loaded ATO builtins/Integers
I: Loaded ATO builtins/Dates
I: Loaded ATO builtins/Booleans
Prelude> _
```

For issuing terms to reduction you just type them in the console prompt. `prosoft-reduce` will reduce them in the context of the current ATO, identified in the command prompt (e.g. `Prelude>` indicates that the current ATO is `Prelude`). If you issue and ICS call, the current ATO does not matter (expect for input variables, as we'll see later).

```
Prelude> ICS(Booleans, not, [false])
=> true
Prelude> _
```

Commands

For issuing commands for the tool, you enter their names after a slash ("/"), like in `/commandname`. Some commands expect parameters, and others don't. The available commands are:

- `/load ATO1 ATO2 ...`
  Loads the ATO's informed as arguments. The ATO's are searched in files

with their same names plus the `.prosoft` extension. `prosoft-reduce` searches them in the current directory, then in a subdirectory `examples`, then in `prosoft-reduce`'s directory for builtin ATO's.

Example (the VideoClub ATO is in a file named `VideoClub.prosoft` inside an `examples` subdirectory):

```
Booleans> /load VideoClub
I: Loaded ATO VideoClub
=> .
Booleans> _
```

- `/in ATO`
  Changes the context of reduction to the ATO informed as argument. You will note that `prosoft-reduce`'s reduce prompt will change to reflect this.

  Example:

```
Prelude> /in Booleans
=> Now in the context of ATO Booleans
Booleans> _
```

- `/help`

  Shows online help.

- `/atos`

  Lists the available ATO's in the environment (all the ATO's that were already loaded).

- `/show ATO1 ATO2 ...`

  Shows operations, variables and equations in the ATO's passed as argument (the ATO's must be already loaded).

- `/match TERM1 TERM2`

  Tests `TERM2` for matching against `TERM1`, as if you were trying to reduce `TERM2` and `TERM1` was the letf-hand side of an equation. If the terms match, the generated substitution is shown.

  Example:

```
Prelude> /match succ(x) succ(pred(succ(y)))
{"x":=pred(succ(y))}
=> yes
Prelude> /match op(x) y
=> no
Prelude> _
```

Input variables

Input variables (`x?`) are an important element of Algebraic Prosoft. They allow us to have more than one input even for monadic operations: when Prosoft stops reducing a term, any remaining input variables are presented to the user, so that he/she can enter an actual value for that variable, allowing the reduction to continue.

Suppose you enter the following term in `prosoft-reduce`:

```
Booleans> and(true,t?)
```

In this case, `prosoft-reduce` can't proceed the reduction without knowing the value of `t?`. It will ask the user for input:

```
... t? ...
t = _
```

After you enter an actual value for `t?`, the reduction can finally be completed:.

```
true<ENTER>
=> true
Booleans> _
```

But `prosoft-reduce` asks for input only when its needed: the value of `t?` is not needed at all when reducing the following terms:

```
Booleans> and(false,t?)
=> false
Booleans> or(true,t?)
=> true
Booleans> _
```

Here are other examples, this time in the "Integers" ATO:

```
Integers> succ(x?)
... succ(x?) ...
x = zero
=> succ(zero)
Integers> multiply(succ(x?),zero)
=> zero
Integers>
Integers> add(succ(succ(x?)),succ(y?))
... add(succ(succ(succ(x?))),y?) ...
x = zero
y = zero
=> succ(succ(succ(zero)))
Integers> _
```

In the end of appendix C, it is presented a sample `prosoft-reduce` session running the examples presented there.

APPENDIX B BUILT-IN ATO'S

This appendix presents the built-in ATO's implemented for `prosoft-reduce`. They are all implemented in Algebraic Prosoft itself.

Booleans

```
specification Booleans
  sort Boolean
  operations
    true : -> Boolean
    false : -> Boolean
    not : Boolean -> Boolean
    and : Boolean, Boolean -> Boolean
    or : Boolean, Boolean -> Boolean

  variables
    v : Boolean
    u : Boolean
  equations
    not(true) = false
    not(false) = true

    and(true,v) = v
    and(false,v) = false
    and(v,u) = and(u,v)

    or(true,v) = true
    or(false,v) = v
    or(v,u) = or(u,v)

end

# vim: tw=60
```

Dates

```
# Dates specification for Prosoft
#
# This specification is currently almost empty, since
```

```
# without a little bit of syntatic sugar is quite difficult
# to express actual dates. Did you already try to write
# "2006" as succ(succ(...succ(zero))) ? ;-)

specification Dates

  include Integers

  sort Date

  operations
    create_date: Integer, Integer, Integer -> Date

end

# vim: tw=60
```

Integers

```
# Integers specification for Prosoft

specification Integers
  include Booleans
  sort Integer
  operations
    zero  : -> Integer
    succ  : Integer -> Integer
    pred  : Integer -> Integer
    add   : Integer, Integer -> Integer
    minus : Integer, Integer -> Integer
    is    : Integer, Integer -> Boolean
    gt    : Integer, Integer -> Boolean
    max   : Integer, Integer -> Integer
    multiply: Integer, Integer -> Integer

    # testing "constructors":
    one   : -> Integer
    two   : -> Integer
  variables
    x : Integer
    y : Integer
  equations

    one = succ(zero)
    two = succ(one)

    pred(zero) = zero
    pred(succ(x)) = x

    add(x,succ(y)) = add(succ(x),y)
    add(x,zero) = x
```

```
33     add(zero,x) = x
34
35     minus(succ(x),succ(y)) = minus(x,y)
36     minus(zero,x) = zero
37     minus(x,zero) = x
38
39     is(succ(x),succ(y)) = is(x,y)
40     is(x,x) = true
41     is(succ(x),zero) = false
42     is(zero,succ(x)) = false
43
44     gt(succ(x),succ(y)) = gt(x,y)
45     gt(succ(x),zero) = true
46     gt(zero,x) = false
47     gt(x,x) = false
48
49     lt(x,y) = gt(y,x)
50
51     max(x,y) = if gt(x,y) then x else y
52
53     multiply(x,zero) = zero
54     multiply(x,succ(y)) = add(x,multiply(x,y))
55
56 end
57
58 # vim: tw=60
```

Lists

```
1  # Lists specification for Prosoft
2
3  specification Lists
4    include Booleans
5    formal sort Component
6    sort List
7    operations
8      emptyList : -> List
9      cons : Component, List -> List
10     head : List -> Component
11     tail : List -> List
12     concat : List,  List -> List
13     reverse : List -> List
14     contains: List, Component -> Booleans
15   equations
16     head(cons(h,t)) = h
17     head(emptyList) = error
18
19     tail(cons(h,t)) = t
20     tail(emptyList) = emptyList
21
22     concat(cons(h,t),l) = cons(h,concat(t,l))
```

```
    concat(emptyList,l) = l
    reverse(cons(h,t)) = concat(reverse(t),cons(h,emptyList))
    reverse(emptyList) = emptyList

    contains(emptyList,x) = false
    contains(cons(h,l),x) = if h==x then true else contains(l,x)
end

# vim: tw=60
```

Maps

```
# Maps specification for Prosoft

specification Maps

  formal sorts Domain, Range
  sort Map

  operations
    emptyMap: Map
    overwrite : Map, Domain, Range -> Map
    lookup: Map, Domain -> Range
    restrict: Map, Domain -> Map
    has_key: Map, Domain -> Map

  variables

    d : Domain
    x : Domain
    r : Range
    m : Map

  equations

    lookup(emptyMap,d) = error
    lookup(overwrite(m,d,r),x) =
      if (d==x)
      then r
      else lookup(m,x)

    restrict(emptyMap,k) = emptyMap
    restrict(overwrite(m,d,r),x) =
      if (d==x)
      then m
      else overwrite(restrict(m,x),d,r)

    has_key(emptyMap,x) = false
    has_key(overwrite(m,d,r),x) =
      if (d==x)
        then true
```

```
40         else has_key(m,x)
41
42 end
43
44 # vim: tw=60
```

Prelude

```
 1 # Prelude specification for Prosoft
 2 #
 3 # This specification contains nothing at all. It’s only a
 4 # placeholder for being the default context in
 5 # prosoft-reduce, where everything needs te be feed into ICS
 6 # to be reduced.
 7
 8 specification Prelude
 9 end
10
11 # vim: tw=60
```

Records

```
 1 # Records "template" specification for Prosoft
 2 #
 3 # WARNING:
 4 #  this file isn’t a valid prosoft specification. Don’t try
 5 #  to pass it as input to prosoft-reduce
 6 #
 7 # This is not an actual specification. Records specification
 8 # are created on demand, since depending on the number of
 9 # fields, the record’s specification is different. This is
10 # only a placeholder file to show how a Record specification
11 # will look like after being created.
12 #
13 # See Builtins.lhs for the actual Records specification
14 # generation
15
16 # assume n = number of fields
17 specification Records
18
19   formal sorts Sort_1, Sort_2, ..., Sort_n
20   sort Record_n
21
22   operations
23
24      create_record : Sort_1, Sort_2, ..., Sort_n -> Record_n
25
26      get_field_1 : Record_n -> Sort_1
27      get_field_2 : Record_n -> Sort_2
28      #...
```

```
     get_field_n : Record_n -> Sort_n

  equations

     get_field_1 (create_record(v_1 ,v_2, ..., v_n)) = v_1
     get_field_2 (create_record(v_1 ,v_2, ..., v_n)) = v_2
     #...
     get_field_n (create_record(v_1 ,v_2, ..., v_n)) = v_n

end

# vim: tw=60
```

Sets

```
# Sets specification for Prosoft

specification Sets

  include Booleans,Lists

  formal sorts Element
  sorts Set

  operations
    emptySet: -> Set
    insert: Element, Set -> Set
    in: Element, Set -> Boolean
    singleton: Element -> Set

    union: Set, Set -> Set
    intersection: Set, Set -> Set

  variables
    e: Element
    x: Element
    s: Set

  equations

    emptySet = set(emptyList)

    insert(e,set(s)) =
      if contains(s,e)
        then set(s)
        else set(cons(e,s))

    in(e,set(s)) = contains(s,e)

    singleton(e) = insert(emptySet,e)

```

```
37     union(set(emptyList),s) = s
38     union(set(cons(x,l)),s) =
39       if in(x,s)
40       then union(set(l),s)
41       else union(set(l),insert(x,s))
42
43     intersection(set(emptyList),s) = emptySet
44     intersection(set(cons(x,l)),s) =
45       if in(x,s)
46         # {x} U (l ^ s):
47         then union(singleton(x),intersection(set(l),s))
48         else intersection(set(l),s) # l ^ s
49
50 end
51
52 # vim: tw=60
```

Unions

```
 1 # Unions "template" specification for Prosoft
 2 #
 3 # WARNING:
 4 #  this file isn't a valid prosoft specification. Don't try
 5 #  to pass it as input to prosoft-reduce
 6 #
 7 # This is not an actual specifications. Unions
 8 # specifications are created on demand, since depending on
 9 # the number of choices, the union's specification is
10 # different. This is only a placeholder file to show how a
11 # Union specification will look like after being created.
12 #
13 # See Builtins.lhs for the actual Unions specification
14 # generation
15
16 # assume n = number of choices
17 specification Unions
18
19   include Booleans
20   formal sorts Sort_1, Sort_2, ..., Sort_n
21   sort Union_n
22
23   operations
24
25      # constructors
26      tag_1 : Sort_1 -> Union_n
27      tag_2 : Sort_2 -> Union_n
28      #...
29      tag_n : Sort_n -> Union_n
30
31      # observers
32      is_tag_1 : Union_n -> Boolean
```

```
33      is_tag_2 : Union_n -> Boolean
34      #...
35      is_tag_n : Union_n -> Boolean
36
37    equations
38
39      is_tag_1(tag_1(x)) = True
40      is_tag_1(y) = False
41      #
42      is_tag_2(tag_2(x)) = True
43      is_tag_2(y) = False
44      #...
45      is_tag_n(tag_n(x)) = True
46      is_tag_n(y) = False
47
48  end
49
50  # vim: tw=60
```

APPENDIX C THE VIDEOCLUB EXAMPLE, REVISITED

To provide a sample of Algebraic Prosoft, we present here a complete example of an problem modelled in Algebraic Prosoft. The following example presents the specification of a control application for a Video Club.

The Video Club example has been historically the classical example for Prosoft specifications. In this appendix, the Video Club example is revisited, this time using the "new" interpretation of Algebraic Prosoft. This example can even be executed in `prosoft-reduce`.

According to the Prosoft paradigm, solution to problems are given by defining ATO's that represent the several data elements involved in the problem.

In the next sections, each one of the ATO's defined for the Video Club problem is presented. Both their graphical representation and their full specification are provided.

The last section shows an example `prosoft-reduce` section running this example.

VideoClub

The first ATO defined is *VideoClub*. Figure C show the graphical representation for the VideoClub ATO.

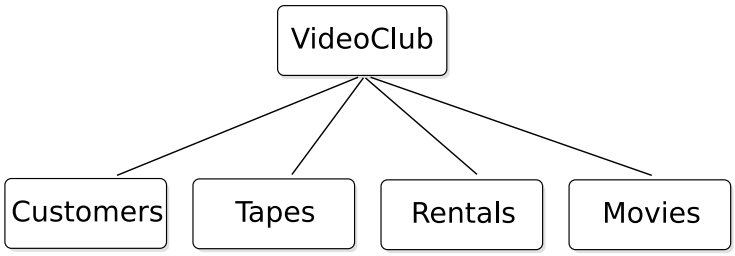


Figure C.1: Graphical representation for *VideoClub* ATO

```
1 specification VideoClub
2
3   include Integers
4
5   include instantiation of Records
6     using
7       Customers for Sort_1,
8       customers for Field_1,
9       Tapes for Sort_2,
```

```
      tapes for Field_2,
      Rentals for Sort_3,
      rentals for Field_3,
      Movies for Sort_4,
      movies for Field_4
  renamed using VideoClub for Record

  operations
    create:  -> VideoClub
    add_customer : VideoClub -> VideoClub
    remove_customer : VideoClub -> VideoClub
    add_movie : VideoClub -> VideoClub
    remove_movie : VideoClub -> VideoClub
    add_tape: VideoClub -> VideoClub
    remove_tape: VideoClub -> VideoClub

    rent_tape: VideoClub -> VideoClub
    return_tape: VideoClub -> VideoClub
    calculate_payment: VideoClub -> Integer

  variables
    c: Customers
    t: Tapes
    r: Rentals
    m: Movies

  equations
    create = create_VideoClub(ICS(Customers,create),
                              ICS(Tapes, create),
                              ICS(Rentals, create),
                              ICS(Movies, create)
                             )

    add_customer(create_VideoClub(c,t,r,m)) =
      create_VideoClub(ICS(Customers,add_customer,[c]),t,r,m)
    remove_customer(create_VideoClub(c,t,r,m)) =
      create_VideoClub(ICS(Customers,remove_customer,[c]),t,r,m)

    add_tape(create_VideoClub(c,t,r,m)) =
      create_VideoClub(c,ICS(Tapes,add_tape,[t]),r,m)
    remove_tape(create_VideoClub(c,t,r,m)) =
      create_VideoClub(c,ICS(Tapes,add_tape,[t]),r,m)

    add_movie(create_VideoClub(c,t,r,m)) =
      create_VideoClub(c,t,r,ICS(Movies,add_movie,[m]))
    remove_movie(create_VideoClub(c,t,r,m)) =
      create_VideoClub(c,t,r,ICS(Movies,remove_movie,[m]))

    rent_tape(create_VideoClub(c,t,r,m)) =
      create_VideoClub(c,t,ICS(Rentals,rent_tape,[r]),m)
    return_tape(create_VideoClub(c,t,r,m)) =
```

```
61        create_VideoClub(c,t,ICS(Rentals,return_tape,[r]),m)
62      calculate_payment(create_VideoClub(c,t,r,m)) =
63        ICS(Rentals,calculate_payment,[r])
64
65  end
```

Customers

The *Customers* ATO describes the base of customers of the Video Club. Figure C show the graphical representation for the the Customers ATO.

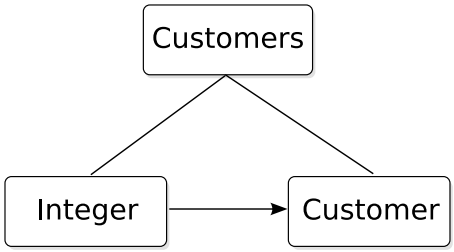


Figure C.2: Graphical representation for *Customers* ATO

```
1  # Videoclub example
2  #
3  # Customers specification
4
5  specification Customers
6
7    include Booleans
8
9    include instantiation of Maps
10     using
11        Integer for Domain,
12        Customer for Reange
13   renamed using Customers for Map
14
15   operations
16
17     create: Customers
18     add_customer: Customers -> Customers
19     remove_customer: Customers -> Customers
20     exists_customer: Customers, Integer -> Boolean
21
22   variables
23
24     cs: Customers
25     customer_code: Integer
26     new_customer: Customer
27
28   equations
```

```
29
30      create = emptyMap
31
32      add_customer(cs) =
33        if not(exists_customer(cs,customer_code?))
34        then overwrite(cs, customer_code?, new_customer?)
35        else cs
36
37      remove_customer(cs) = restrict(cs, customer_code?)
38
39      exists_customer(cs, customer_code) =
40        has_key(cs, customer_code)
41
42  end
43
44  # vim: tw=60
```

Tapes

The *Tapes* ATO describes how are organized information about the tapes owned by the Video Club. The graphical representation on figure C helps in understanding the structure of this ATO.

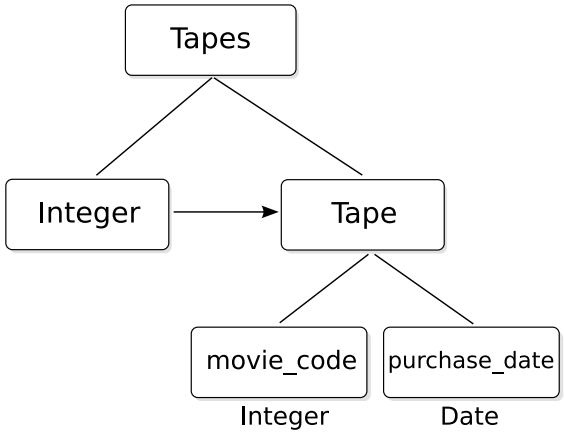


Figure C.3: Graphical representation for *Tapes* ATO

```
1  specification Tapes
2
3    include Integers, Dates, Booleans
4
5    include instantiation of Maps
6      using Integer for Domain,
7            Tape for Range
8    renamed using Tapes for Map
9
10   include instantiation of Records
11     using
12       movie_code for Field_1,
```

```
      Integer for Sort_1,
      purchase_date for Field_2,
      Date for Sort_2
  renamed using Tape for Record

  operations

    create: -> Tapes
    add_tape: Tapes -> Tapes
    remove_tape: Tapes -> Tapes
    exists_tape: Tapes, Integer -> Boolean

    #exists_tape_of_movie: Tapes, Integer -> Boolean

  variables

    ts: Tapes
    t: Tape
    cod: Integer
    mcod: Integer
    purchase_date: Date

  equations

   create = emptyMap

   add_tape(ts) =
     if not(exists_tape(ts, cod?))
     then overwrite(ts, cod?, create_Tape(mcod?, purchase_date?))
     else ts

   remove_tape(ts) = restrict(ts, cod?)

   exists_tape(ts, cod) = has_key(ts, cod)


end
```

Rentals

Rentals is the ATO that represents information about rentals made by customers: the tapes rent, together with a record of the date of the rental.

```
specification Rentals

  include Integers, Dates, Booleans

  include instantiation of Maps
    using
          tape_code for Field_1,
          Integer for Sort_1,
```

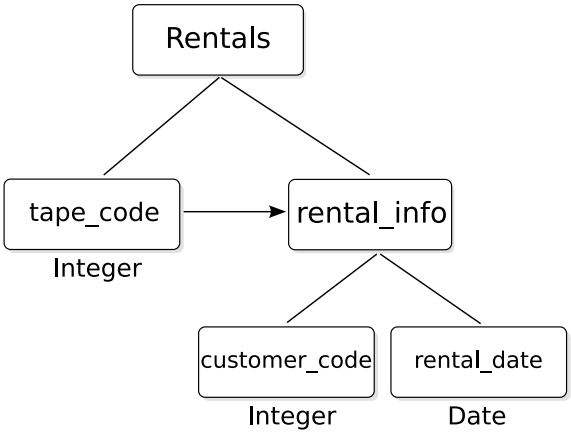


Figure C.4: Graphical representation for *Rentals* ATO

```
9           RentalInfo for Sort_2,
10          rental_info for Field_2
11    renamed using Rentals for Map
12
13    include instantiation of Records
14      using
15          customer_code for Field_1,
16          Integer for Sort_1,
17          rental_date for Field_2,
18          Date for Sort_2
19    renamed using RentalInfo for Record
20
21    operations
22
23      create: Rentals
24      rent_tape: Rentals -> Rentals
25      return_tape: Rentals -> Rentals
26      customer_has_rental: Rentals -> Boolean
27      is_tape_rent: Rentals -> Boolean
28      calculate_payment: Rentals -> Integer
29      diary_tax: Integer
30
31    variables
32      rs: Rentals
33      tape_code: Integer
34      rental_date: Date
35      today: Date
36      customer_code: Integer
37      searched_customer_code: Integer
38
39    equations
40
41      create = emptyMap
42
43      rent_tape(rs) =
```

```
44        if not(is_tape_rent(rs, tape_code?))
45        then overwrite(rs,
46                       tape_code?,
47                       create_rentalinfo(customer_code?, rental_date?)
48                      )
49        else rs
50
51      return_tape(rs) = restrict(rs, tape_code?)
52
53      customer_has_rental(overwrite(rs,
54                                    tape_code,
55                                    create_rentalinfo(customer_code,
56                                                      rental_date)
57                                   )
58                          ) =
59        if (customer_code == searched_customer_code?)
60          then true
61          else customer_has_rental(rs)
62      customer_has_rental(emptyMap) = false
63
64      is_tape_rent(rs, tape_code) = has_key(rs, tape_code)
65
66      calculate_payment(rs) =
67        if is_tape_rent(rs, tape_code?)
68        then multiply(minus(today?,
69                            get_rental_date(lookup(rs, tape_code?))
70                           ),
71                      diary_tax
72                     )
73        else zero
74
75      diary_tax = succ(succ(zero)) # 2/day
76
77  end
```

Movies

The *Movies* ATO represents information about the movies the Video Club know about. Figure C show the graphical representation of the ATO.

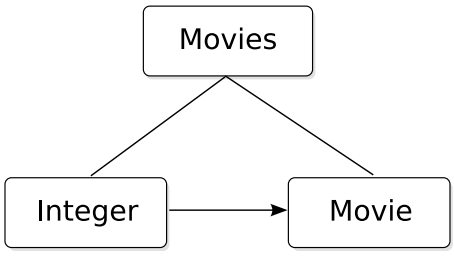


Figure C.5: Graphical representation for *Movies* ATO

```
specification Movies

  include Booleans

  include instantiation of Maps
    using
      Movie for Range,
      Integer for Domain
  renamed using Movies for Map

  operations

    create: Movies
    add_movie: Movies -> Movies
    remove_movie: Movies -> Movies
    exists_movie: Movies -> Boolean

  variables

    ms: Movies

  equations

    create = emptyMap

    add_movie(ms) =
      if not(exists_movie(ms, movie_code?))
      then overwrite(ms, movie_code?, new_movie?)
      else ms

    remove_movie(ms) = restrict(ms,movie_code?)

    exists_movie(ms, movie_code) = has_key(ms, movie_code)

end
```

Customer

The *Customer* ATO represents information the Video Club stores about each customer. Figure C shows the graphical representation for the ATO.

```
# Videoclub example
#
# Customer specification

specification Customer

  include instantiation of Records
    using
      String for Sort_1,
      Name for Field_1,
```

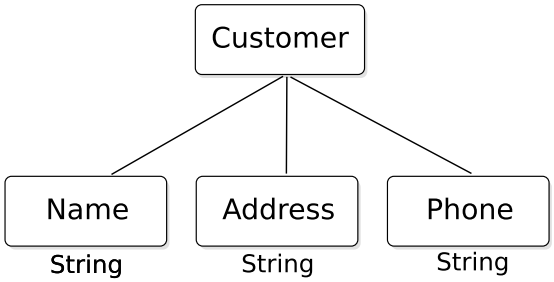


Figure C.6: Graphical representation for *Customer* ATO

```
      String for Sort_2,
      Address for Field_2,
      String for Sort_3,
      Phone for Field_3
  renamed using Customer for Record

end

# vim: tw=60
```

Movie

The *Movie* ATO represents informations that the Video Club stores about each movie. Graphical representation for the ATO is shown in figure C.

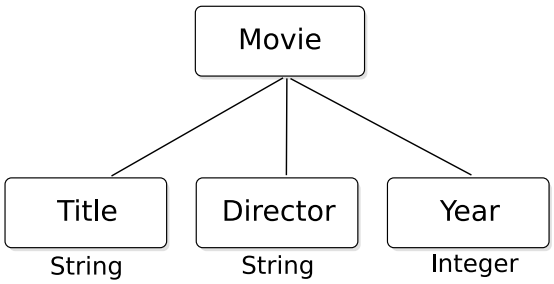


Figure C.7: Graphical representation for *Movie* ATO

```
specification Movie

  include instantiation of Records
    using
      String for Sort_1,
      Title for Field_1,
      String for Sort_2,
      Director for Field_2,
      Integer for Sort_3,
      Year for Field_3
  renamed using Movie for Record

end
```

Running a sample `prosoft-reduce` session with the VideoClub example

First we start `prosoft-reduce`:

```
# prosoft-reduce
I: Loaded ATO builtins/Prelude
I: Loaded ATO builtins/Sets
I: Loaded ATO builtins/Maps
I: Loaded ATO builtins/Lists
I: Loaded ATO builtins/Integers
I: Loaded ATO builtins/Dates
I: Loaded ATO builtins/Booleans
```

Then we load all the needed ATO's:

```
Prelude> /load Customers Movie Rentals Customer Movies VideoClub
    Tapes
I: Loaded ATO Tapes
I: Loaded ATO VideoClub
I: Loaded ATO Movies
I: Loaded ATO Customer
I: Loaded ATO Rentals
I: Loaded ATO Movie
I: Loaded ATO Customers
=> .
```

Creating a new VideoClub object:

```
Prelude> ICS(VideoClub,create)
=> create_VideoClub(emptyMap,emptyMap,emptyMap,emptyMap)
```

Now we add a new customer:

```
Prelude> ICS(VideoClub, add_customer, [create_VideoClub(emptyMap
   ,emptyMap,emptyMap,emptyMap)])
... create_VideoClub(overwrite(emptyMap,customer_code?,
   new_customer?),emptyMap,emptyMap,emptyMap) ...
customer_code = succ(zero)
new_customer = john
=> create_VideoClub(overwrite(emptyMap,succ(zero),john),emptyMap
   ,emptyMap,emptyMap)
```

And then another one:

```
Prelude> ICS(VideoClub,add_customer,[create_VideoClub(overwrite(
   emptyMap,succ(zero),john),emptyMap,emptyMap,emptyMap)])
... create_VideoClub( if not( if succ(zero)==customer_code? then
    true else false) then overwrite(overwrite(emptyMap,succ(zero
   ),john),customer_code?,new_customer?) else overwrite(emptyMap
   ,succ(zero),john),emptyMap,emptyMap,emptyMap) ...
```

```
customer_code = succ(succ(zero))
new_customer = mary
=> create_VideoClub( if not(false) then overwrite(overwrite(
   emptyMap,succ(zero),john),succ(succ(zero)),mary) else
   overwrite(emptyMap,succ(zero),john),emptyMap,emptyMap,
   emptyMap)
```

Here we get something weird. Since we were in the context of Prelude (an empty ATO), it wasn't possible to reduce `not(false)`. If we just jump to the "Booleans" ATO (or any other ATO that includes it), we get the right result:

```
Prelude> /in Booleans
=> Now in the context of ATO Booleans
Booleans> ICS(VideoClub,add_customer,[create_VideoClub(overwrite
   (emptyMap,succ(zero),john),emptyMap,emptyMap,emptyMap)])
... create_VideoClub( if not( if succ(zero)==customer_code? then
    true else false) then overwrite(overwrite(emptyMap,succ(zero
   ),john),customer_code?,new_customer?) else overwrite(emptyMap
   ,succ(zero),john),emptyMap,emptyMap,emptyMap) ...
customer_code = succ(succ(zero))
new_customer = mary
=> create_VideoClub(overwrite(overwrite(emptyMap,succ(zero),john
   ),succ(succ(zero)),mary),emptyMap,emptyMap,emptyMap)
Booleans>
```