SPECIFICATION AND IMPLEMENTATION OF THE LARCH SHARED LANGUAGE

Yvonne Everett

A thesis submitted to the Faculty of Science of the University of Cape Town in fulfillment of the requirements for the degree of Master of Science.

Cape Town, 1989

ACKNOWLEDGEMENTS

I would like to express my gratitude to the following people for their assistance:

Professor Ken McGregor, Department of Computer Science, University of Cape Town.

Mr Jeff Dooley, System Technologies (Pty) Ltd, Pinelands.

CONTENTS

Chapter			Page
1.	**Introduction**		1
2.	**Formalism**		5
	2.1	Formal Methods	5
	2.2	Specification Languages	12
3.	**The Larch Shared Language**		23
	3.1	The Larch Project	23
		3.1.1 The Shared Language	27
		3.1.3 The Interface Language	34
	3.2	Design and Specification	37
		3.2.1 Text, Symbols and Lexical Analyzer	46
		3.2.2 Data Structures	49
		3.2.3 Parser	59
		3.2.4 Context Sensitive Checking	67
		3.2.5 Implementation Details	69
4.	**The Semantic Checker**		71
	4.1	Theorem Proving	71
		4.1.1 Substitution and Unification	72
		4.1.2 Term Rewriting Systems	75

4.1.3 Uniform Termination 78
4.1.4 Confluence 92
4.1.5 Proving Theorems 98
4.1.6 Short-Cuts and Synthesis 99

4.2 Design and Specification 100
4.2.1 Substitution, Unification & Reduction 102
4.2.2 Uniform Termination
4.2.3 Confluence and Critical Pairs 109
4.2.4 Semantic Checking 114
4.2.5 Implementation Details 125

5. **Conclusions and Areas for Further Research** 127

5.1 The Larch Shared Language 128
5.2 Specification Languages 131
5.3 Formal Methods based on Specification Languages 134

References

Appendix

A Larch Shared Language Reference
B Examples
C Larch and Prolog

CHAPTER 1 : INTRODUCTION

The Problem

The "software crisis" has been well documented over the last few decades and takes many forms: from early concerns of mounting maintenance problems; to concerns of productivity - comparing the phenomenal increases in hardware capability to the software bottleneck; to more recent concerns of software liability suits [Economist 88] and risks to the public [Neumann 88].

Following Aristotle's example, Brooks separates the essence of the problem (inherent difficulties in the nature of software: complexity, conformity, changeability and invisibility) from accidents (difficulties that revolve around the production of software today, but are not inherent). [Brooks 87]

The problem is essentially one of managing complexity on a vast scale in a fast changing environment.

Past progress has helped solve the problem of productivity, eg high-level languages, timesharing, unified programming environments such as Unix and sophisticated workstations. Future progress lies in addressing the essence of the software problem, in particular the areas of requirements refinement, rapid prototyping and incremental development.

One approach to this problem is formal methods centered on a

formal specification language. The Larch Project at MIT is concerned with developing languages, tools and techniques for formal specification [Guttag, Horning and Wing 85].

Objective

This project aims to prototype formal specification in Larch.

The motivation for looking at formal specifications stems from an appreciation of the problem outlined above, frustration with current methods, and a desire to practise what is preached. The aim is to implement a formal specification language, to write a non-trivial specification and to employ formal methods of specification during software development. As a result, one should have a thorough understanding of a formal specification language, and the practical implications of using it as a basis for formal methods.

Larch was selected for various reasons. It is a relatively mature specification project with some years of research behind it (earliest work dates from 1975). There exists extensive literature on Larch, and complete documentation on the language was available, including a handbook of examples and a detailed reference containing the grammar and semantic checks. No literature, however, could be found on projects which had actually been developed using Larch specifications and formal methods. Larch is centered on the algebraic method of specification, a method which I favoured and had some prior experience in.

Methods

There are three inter-related aspects to the project:

* Implementing the Larch Shared Language.
  The main academic work here involved developing methods for performing the complex semantic checks required. This required a theorem prover.
  The implementation of Larch was then used to check a specification.

* Writing a non-trivial specification.
  A specification was written for this implementation of a parser and the semantic checker for the Larch Shared Language itself.
  This specification was used as data for testing the implementation. The experience gained was subsequently used to evaluate the Larch Shared Language, with inferences for specification languages in general.

* Employing formal methods of specification.
  The method of specifying the problem (in Larch) before programming (in C) the parser and semantic checker was followed. The experience gained was used to evaluate formal methods centered on formal specifications.

The second chapter of this report presents the case for formal methods and describes various specification languages and systems.

The third chapter provides an introduction to the Larch

Project, and outlines the requirements for semantic checking. A specification for the Larch Shared Language parser and context-sensitive checks is given, followed by details of the implementation.

The fourth chapter describes the method of theorem proving chosen (Term Rewriting Systems) to form the basis of the semantic checker. The methods of performing the semantic checks are then detailed, together with the specification and details of the implementation.

The final chapter presents the conclusions for the three areas of study: Larch, specifications in general, and formal methods.

CHAPTER 2 : FORMALISM

2.1 FORMAL METHODS

Definition

"Formal" is defined variously as: pertaining to the form or constitutive essence of a thing; logic, concerned with the form as distinguished from the matter of reasoning; explicit and definite as opposed to tacit; marked by excessive regularity and symmetry, stiff or rigid in design; having a set or rigorously methodical character. Not only do dictionary definitions of "formal" differ, but also those among computer scientists: Larch may be considered a formal specification language when compared to English, yet it is informal in that its semantics have not been formally defined.

Formal methods offer guidelines, techniques, tools and forms of organization. They are usually centered around a formalism such as a language with formal syntax and semantics. A distinction should be made between a global method supporting software development as a whole, and components of a method which support specific tasks. For any formal component to be useful, it must be carefully embedded in the whole process, since the process as a whole will determine the quality of the product. [Floyd 85]

Formal methods are seen as either a universal panacea, or a necessary evil. The arguments for both positions, with special emphasis on specification, are given, and then summed up.

Arguments for Formal Methods and Specification

Formalists take a **product-oriented** perspective on software development [Floyd 88]. Formalists argue that the task of software development is to transform fixed requirements into a correct program in several steps - specify in abstract terms what is to be done, and then derive the program.

The software developer is "outside" the environment the system is being developed for, the environment is essentially static, and the effect of the software on the environment is predictable.

Formal methods inspire confidence in a system, and lead to a better quality design. Confidence and quality are the result of consistent application of a tested development method. Quality is associated with the features of the product (reliablity, efficiency), is determined by validation (testing, proving) and is defined by looking from the program to the user (eg user friendliness, acceptability).

Formalization early in the process, at specification time, enables design errors and inconsistencies to be detected when correcting them is still cheap and simple. Formal specifications provide a clear and concise model of a system prior to construction. Questions about the functionality of

the design can be answered at an early stage by referring to the specification, or if the specification is executable, it can serve as a prototype.

The meaning of the program is defined by its formal semantics given in the specification, the relation between the specification and the real world is left open. The specification forms a contract between the user and the developer; and serves as a communication tool between members of a team. Programs are understandable from documents only. Knowledge of the system is acquired by understanding these documents, thus it is vital that they are complete, consistent and unambigious, and preferably only use one formalism. Software tools may relieve the programmer of some of the documentation work, and help make changes conveniently and consistently in all documents.

The power of formal languages is to enable specifications to be mechanically checked, the transition to code to be partially automated, and the code subjected to generated tests or mechanically verified. Programs can be proved correct with respect to the specification.

Arguments against Formal Methods and Specification

The contrary perspective is **process-oriented** [Floyd 88]. Software development is an iterative process of learning, cooperation and communication, the object being not so much to produce a product from fixed requirements, but a change in the work processes of the humans who use the product.

Environments are never static, but dynamic and evolving. The software developer should become part of the environment since the processes of development and use affect each other.

There can be no formal method for software development as a whole, as other issues are involved which cannot be formalized. Some requirements can be formalized in advance, but not the changing aspects that determine the usability of the product as a tool for humans: handling of errors, the matching of system functions to work steps, the distribution of functions between man and machine.

Adequacy for the task is a better criteria for measuring quality than correctness. Quality should be defined from the users perspective (relevance, suitability, adequacy) and determined by evaluation (trial use, critical appraisal) not by validation.

Software design is a creative, intuitive process [Naur 85]. An initial vision is gradually refined, and formalism serves only to hinder this process.

All formal specifications are rooted in common understanding, for example algebraic notation. All software discussions rest on a common tacit understanding of the application context. For any formal document to be useful, we have to ensure that readers relate to our tacit understanding, by giving the context or decisions that led to the design. The relation of the specification to the real world must be clarified. Knowledge of a program is acquired through trial use and discussions, and documents should facilitate this. Documents should describe the program by example and analogy, and from

defined viewpoints in language understandable to the readers.

Formal methods fail to address key issues such as finding out the relevant requirements and their interactions; designing a system taking into account both the underlying machine and the informal context of the system's use as a human tool; adapting existing systems to meet new needs.

Formal Methods and Specification in Perspective

Reality probably lies somewhere between these two extremes, a balance between process and form needs to be maintained at all levels of software development. The two viewpoints should be seen as complementary, not contradictory. Their relative importance will depend on the type of system and its usage, for example compilers are more amenable to formal approaches than interactive application systems which stand to gain from rapid prototyping.

One needs to examine the relevance of the formalist standpoint : are its assumptions valid and its claims justified? Not all arguments for (or against) formalism carry equal weight in all situations.

The usability of any underlying formal method is dependent on the convenience and usability of its tools. Specialized training may be required in its use, since understanding specifications often involves understanding abstract mathematical properties. Formal methods may assist in formulating the functions of the system, but they are no aid in understanding how the humans developing the system, or the

humans using the system, function.

Specifications focus on correctness and leave aside issues such as choosing a good design, yet the choice of the abstraction has an impact on clarity and design.

Formal specifications are not the only method of finding errors and omissions - a simple checklist may be just as useful. Nor can formalism pick out discrepancies between the real needs, and the formulated requirements.

Program proofs rest on argumentation - a social human process subject to the same errors as the program [DeMillo, Lipton, Perlis 79]. Proving a program correct with reference to its specification may be meaningless, since the specification is a formal version of an unreliable communication process early on in software development. Techniques are still needed to check that the specification meets the user's needs.

Although there are successful cases where formal specifications have been used on a large scale, there are few cases where programs have been directly derived from the specification, possibly because large parts of the program are beyond the scope of the method - such as storage mechanisms, man-machine interfaces, communications with peripheral devices.

Specifications should not be purely mathematical documents geared to theorists, but aids in communication. There is a need to improve the style and accessibility of formal documents. What is needed are "notions not notations", as Gauss phrased it.

Formal methods are useful to clarify underlying concepts - eg abstract data types. They are useful where requirements are stable, and functionality and reliability are more important than usability, eg in concurrency. They can serve to standardize solutions to well understood problems.

The processes of learning, cooperation and communication should not be formalized, for example the design should be topdown, but not necessarily the process.

By combining the two approaches, two aims of software development can become achievable - building the product right, and building the right product.

2.2 SPECIFICATION LANGUAGES

> "The most useful function that the software builder performs is the iterative extraction and refinement of the product requirements: for the truth is the client does not know what he wants." [Brooks 87 p17]

Specifications (even informal ones) are widely accepted to be useful in organizing ideas, documenting design decisions and as a basis to compare alternative designs.

Many varied specification languages have been developed to overcome the "Seven Sins" of specifying - noise (redundancy), silence (omissions), over specification, contradiction, ambiguity, forward reference, wishful thinking; and to avoid the weaknesses of natural language [Meyer 85 p7].

Formal specifications are written in a language with a precisely defined syntax and semantics. This precision is essential in providing machine support to reason about specifications.

An abstract specification not oriented towards any implementation may be used as a basis in design and implementation. A designer can then use and reuse this module without knowledge of its implementation, while the programmer can implement it without knowing its use(r)s.

Principles of good Specifications

[Balzar & Goldman 86] outline the principles of a good specification and their implications for specification language design. Good specifications separate functionality from implementation. They specify the system of which the software is a component and the environment in which the system operates. They provide notations to specify entities which behave as processes (mathematical functions) and the dynamic environment in which the entities interact. In addition the specification should be a conceptual model describing the system as perceived by the users. It needs to be operational, but tolerate incompleteness. It must be localized and loosely coupled to allow modifications.

Using these principles as a basis, Balzar and Goldman outline an ideal specification langauge which includes a global relational data model, with inference ability. Larch does not meet their (possibly unachievable) requirements, but still represents some progress in this direction.

Types of Specifications and Specification Languages

Specifications can be classified by size (small or large), viewpoint (language oriented or application oriented), or constraint (constrain the behaviour of the implementation or constrain the structure) [Guttag, Horning and Wing 85b p32]. In terms of this we can define

* System specifications which are application-oriented, behavioural specifications of typically large programs, with constraints expressed in terms of what the user observes.
* Local specifications which are language oriented, behavioural specifications on typically small program units, with constraints on the program in terms of the programming language, for example Larch interface specifications.
* Organizational specifications which combine structural and behavioural specifications of the components. An organizational specification satisfies a system specification if each component satisfies its own specification.

[Liskov & Berzins 86] divide specification languages into two classes by abstraction:

* Procedural Abstraction, such as Hoare's I/O approach which uses assertions and has ease of verification as its main benefit.
* Data Abstraction centers on the data type and operations on it. In the axiomatic style of Larch, all objects are produced by constructor operations on that type with

enquiry operations to extract information.

In both classes, proof that an implementation meets a specification is equally difficult - one must either show a homomorphism or show that the axioms are satisfied. The complexity of the specification in both classes depends on the complexity of the abstraction.

Yet another classification [Wing 87] defines

* the operational approach, where one gives a method of constructing a program or data type, such as state machines [Parnas 72], useful for defining user interfaces [Jacob 83] or communication protocols [Sunshine et al 86]; or processes in Paisley [Zave & Schell 86] for defining realtime systems. The specifications are often executable, such as those written in Gist [Balzar 85].
* the definitional approach, where the properties of the abstract type are specified, not the method of constructing it. Both the axiomatic style of Larch interface specifications and the algebraic style of the shared language traits belong here. Other similar languages include Clear [Burstall & Gougen 86].

Larch differs from other languages in its two-tiered approach, in particular in using the interface tier to provide a simple way of dealing with errors and side-effects. Unlike other definitional languages (such as Clear), the semantics are defined in terms of theories, ie sets of first-order formulae, and not in terms of initial or final algebras. Larch is one of the few languages to provide a simple way of specifying errors and side effects.

These classifications deal only with specification languages for sequential programs, more work is needed in the area of concurrency.

Evaluating specifications

Specification approaches can be evaluated on the following criteria [Zave 88]:

* Relation to requirements - can all the requirements be specified?
* Human factors - is the specification comprehensible, how much intelligence, knowledge and effort is required to write (and read) them, is there guidance (such as procedures and heuristics) for writing them, do they enhance communication?
* Quality and Production Benefits - can they be checked for consistency, are they easily modified, is there aid in determining implementation strategies and implementation, can one ascertain whether the implementation satisfies the specification, is there aid in documentation and the keeping of project histories?
* Availability - Are the techniques and tools available?

Key questions to be asked about specifications include [Guttag, Horning and Wing 85b p26]:

* What is accomplished by writing them?
* Who should write them?
* When should they be written?

* What benefits result from their existence?
* Who should read them?
* Which properties should be used to evaluate them?
* What should be done with them afterwards?

Answers to these questions can differ for different types of specifications.

Mechanized Support

Specification languages need support tools such as:

* Libraries to enable reuse of existing specifications, and cross-referencers to trace specifications
* Paraphrasers and summarisers to help overcome the tradeoff between rigor and understandability,
* Methods for producing semantically consistent but notationally distinct representations,
* Language-sensitive editors for incremental construction,
* Theorem provers to check the desired properties hold,
* Interpreters to provide a means for checking through examples,
* Symbolic evaluation to help explore entire classes of test cases simultaneously, providing a dynamic means for validating a specification,
* Verifiers to help demonstrate that an implementation satisfies the specification.

These tools should be included in an integrated specification system, what [Balzar 83] terms an "automated Knowledge-based Assistant".

The Software Lifecycle Redefined

Ideally specification languages should not only be embedded in a specification system, but form the basis of the software lifecycle.

The traditional software lifecycle is the Waterfall Model: a linear process of refinement from informal specification to

program - a highly detailed formal object. Prototyping is uncommon, implementation is manual, and maintenance is an afterthought.

The Waterfall approach has two fatal flaws [Balzar 83]:

* there is no technology for managing knowledge-intensive activities. The process of design, converting a specification into an efficient implementation is informal, people-intensive and undocumented. Yet it is this information that is needed for maintenance.
* Maintenance is performed on source code, where information has been distributed and hidden by optimization, thus making the system harder to understand. Maintenance is an onerous task reserved for juniors.

The approach separates behavioural considerations from structural ones which probably accounts for the use of informal English specifications.

A new lifecycle model is needed - one based on automation.

Balzar suggests one that is formal, computer assisted and centered on specifications. Formal specification becomes the standard, the specification is executable and serves as a prototype which can be evaluated. Implementation is machine aided and testing reduced as validity is guaranteed by process, rather than by proof: correctness by design. Documentation is automatic and maintenance is done at specification level and the system re-implemented with computer assistance. Modifications are thus performed at the

level closest to the users conceptual model. Since the specification is operational, the user becomes the analyst, and the analyst's role changes to ensure that the system matches the users intent. By embedding formal specifications in a life-cycle model "more uniform interfaces between the various stages of software development can be achieved".

[Zave 84] suggests a similar operational lifecycle also centered on executable specifications and the separation of problem oriented concerns from implementation oriented concerns. The Paisley [Zave & Schell 86] language is supported by a set of tools including a parser, cross-referencer, interactive interpreter and consistency checker.

Practical Experience with Specification Languages

Theoretical interest in formal specification has been increasing with a growing convergence among formal methods [Horning 85]. Formalisms are being used on a larger scale as languages such as Larch reach maturity [Woodcock 88 p30]. There are three areas of interest: applicability, industrialization and experiences.

Applicability

Consideration must be given to the kinds of problems we can address with formal methods. One reason for the lack of success with formal specifications could be due to an attempt to make a single language serve many purposes. Larch attempts to overcome this through its two-tiered approach. On the

other hand, a single specification language may prove inadequate for all dimensions of the problem: cost, performance, behaviour, reliability, concurrency etc. A solution is to combine formal methods and specification styles, for example using Larch to describe abstract data types and Paisley to describe real-time processes. The catch lies in combining proof techniques.

Other reasons for slow acceptance of formal specifications could be the inaccessibility of theoretical results to practitioners, or the fact that the skills required to write specifications are not widely available. In addition, there are more established alternatives in human to human communication for specifying, than in human to machine communication. Specification systems have come with very little computer support for alternative presentation - overviews, summaries etc.

A gap exists between formal specifications and program development - examples of specifications in the literature do not usually detail how the program is developed from the specification. [Maibaum et al 85] outline an approach to specifying motivated by the practice of programming. Their approach is centered on specifications which resemble Larch "traits".

Industrialization

Formal specifications have been slow to catch on in industry. Real examples with practical benefits are needed to motivate more widespread use. Stacks convince no-one! Training and support are needed for project teams. Suggestions for

approaches when resources are limited are required.

Experiences

Do the advocates of formal specification practise what they preach, and does it work? None say it failed, neither do they claim to have found a universal panacea [Woodcock 88]. Often other factors are involved in a successful project: well motivated and trained people and supportive management, and a well-chosen application.

[Berry & Wing 85] observed a number of successful projects, some developed by specifying, others by prototyping. They note that success may be due to the second-time phenomenon: the success is due to a formal, machine checked first pass, and has less to do with the method or language chosen for the first pass. This implies that informal specifications or haphazard prototyping may not be successful.

[Gehani 86] gives results of a comparison between an informal English specification of an existing, successful system and a formal specification he wrote. Surprisingly, it brought to light two flaws in the system which hadn't been discovered during the original specification process or the implementation. Perhaps specification can play a role after the event?

CHAPTER 3 : THE LARCH SHARED LANGUAGE

3.1 THE LARCH PROJECT

The Larch Project [Horning 85] [Guttag, Horning & Wing 85] [Wing 87] at MIT's Laboratory for Computer Science is developing tools and techniques intended to aid in the productive use of formal specifications. It is based upon a two-tiered approach to specification, separating the specification of state transformations from the underlying abstractions. Thus each specification has components written in two languages, one designed for a specific programming language, and another common to all languages. The former are the Larch interface languages, the latter is the Larch Shared Language with which this project is concerned.

Assumptions

The direction the MIT Larch project has taken has been influenced by assumptions made concerning specifications:

* that behavioural specifications of components of sequential programs could be useful;
* that specifying is as error-prone as programming and therefore specifications should be readable to facilitate human checking, yet formal and incorporate redundancy to facilitate mechanical checking;

* it is essential that large specifications are composed of smaller ones in order to handle complexity, ie the "putting together" operations are crucial;
* specifications are constructed incrementally with irrelevant details often delayed, it is essential to be able to reason about incomplete specifications and distinguish between oversights and intentional incompleteness;
* many abstractions can be defined independently of a programming language, for others it is impossible to avoid bias.

Main Features

As a result of these assumptions the Larch family of languages has the following features:

* specifications are reusable, and a handbook of common specifications is available [Guttag, Horning & Wing 85]
* composability to support incremental construction from existing specifications (ie. a specification can assume, import, or include other specifications);
* emphasis is laid on presentation, the specifications are designed to be readable, with the composition mechanisms operating on specifications rather than on theories or models;
* comprehensive checks built into the language assume the availability of a powerful theorem prover;
* the Shared language has a simple semantic basis taken from algebra;
* while the Interface languages are based on predicate calculus, with semantics defined in terms of a

programming language. This means errors, side-effects and resource allocation are dealt with using language specific notations.

Larch is notable in its efforts to keep it simple, yet powerful. For example, the "without" facility common in other specification languages was dropped as it was felt that any specification including without could be written in another way.

> "Everything should be as simple as possible, but no simpler." (A. Einstein)

The Two-Tiered Approach

The Larch project adopted a two-tiered methodology : each Larch specification has one component written in the Larch Shared language, and another component written in an interface language. The shared language is independent of any programming language, whereas the interface languages are derived from programming languages.

The advantages of the two-tiered approach lies in the separation of concerns - the underlying abstractions are separated from the state transformations. It is hoped that this approach will encourage the use of specification languages in practice, since the separation of the two concerns results in languages that are accessible to both designers and programmers. The division of effort also means a designer can specify modules separately without concern for their implementation, and the programmer implement them separately without concern for their use. A consequent

advantage is increased portability - well defined traits can form the basis for specifications in different interface languages as well as different applications.

The term "Shared" is used since all interface specifications rely on the same language to define abstractions. The term "interface" is used since interface specifications define only the observable behaviour of a module. The shared language defines terms used in interface specifications - these terms form the link between the two tiers.

Larch encourages a style of specification where most of the structural complexity is pushed into the shared language and data abstraction plays a central role. As a result, the shared language has mechanisms for combining specifications, and for checkable redundancy, whereas the interface languages do not.

The ideas behind Larch are probably more important than the details - a useful method is more than a collection of good ideas.

3.1.1 THE SHARED LANGUAGE

"Traits" are the basic unit of specification in the Shared Language. A trait defines an abstract data type in terms of the operations that can be performed on it. The abstract data type is called the "sort of interest". Larch uses the words "operators", "sort", and "term" to avoid confusion with the similar concepts "function", "type" and "expression" which are commonly found in programming languages.

Operators on the sort of interest can be separated into

* constructors: the sort is "generated" by these operators,
* observers: these are usually "convertible". The sort is usually "Partitioned" by the observers.

The operators are constrained by equations that relate terms containing them. A good heuristic for generating enough equations to adequately define the abstract data type is to write an equation for each observer applied to each constructor.

Examples

These examples of specifications and their associated theories illustrate the main features of the Larch Shared Language. The syntax and semantics of the language are more fully described in appendix A.

The trait **Container** abstracts common properties of data structures that contain elements such as sets, stacks, queues.

```
Container : trait
introduces
     new       :          ->   C
     insert    : C, E     ->   C
constrains C so that C generated by [new, insert]
```

The part of the specification after **introduces** declares the operators, each with its signature, ie the sort of its domains and range. **C** is the sort of interest, **new** and **insert** are both constructors, as is indicated by the **generated by** clause.

The trait **IsEmpty** assumes **Container** and **constrains** the operators it inherits, adding checkable redundancy with the **converts** clause. This means that any term with no variables of sort C should be provably equal to one that does not contain **isempty**. Because of the **generated** clause in **Container**, this can be proved by induction using **new** as a basis and **insert(c,e)** in the induction step.

The standard traits **boolean**, **equality** and **ifthenelse** are imported into all traits if needed. **Boolean** would be imported into **IsEmpty** to provide the sort **Bool**, and the operators **true** and **false**.

```
IsEmpty : trait
assumes Container
introduces IsEmpty : C -> Bool
constrains isempty, new, insert
so that for all [ c : C, e : E]
     isempty(new) = true
     isempty(insert(c,e)) = false
```

```
implies converts [isempty]
```

Likewise, **Next** and **Rest** are built upon **Container**, with **exempts** adding checkable redundancy.

```
Next : trait
assumes Container
introduces next : C -> E
constrains next, insert so that for all [e : E]
     next(insert(new,e)) = e
exempts next(new)

Rest : trait
assumes Container
introduces rest : C -> C
constrains rest, insert so that for all [e : E]
     rest(insert(new,e)) = new
exempts rest(new)
```

Enumerable specifies properties common to **containers** that keep their contents in an order, such as stacks. The **partitioned by** clause indicates that **isempty**, **next** and **rest** are sufficient to distinguish between unequal terms of sort C.

```
Enumerable : trait
includes Container, Next, Rest, IsEmpty
constrains C so that C partitioned by [next,rest,isempty]
```

Stack then specializes **Enumerable** making use of the **with** clause.

```
Stack : trait
includes Enumerable with [stk for C, top for next, pop
for  rest, push for insert ]
constrains push, pop, top so that for all [s : stk, e :
E]
     top(push(s,e)) = e
     pop(push(s,e)) = s
```

Notice that no mention is made of routines that operate on stacks nor error handling - this is dealt with in the interface specification. Neither has anything been said about how the stack is to be represented nor the algorithms to manipulate it - this is dealt with in the implementation.

Traits are the principal reusable unit, other **containers** can easily be defined using **Enumerable.**

Theories

The theory associated with each trait is an inference closed set of well formed formulas of typed first order calculus with equality, induction and reduction (see appendix A for a full definition). The theory associated with **Stack** includes the theories associated with the traits it **assumes**, **includes** and **imports.**

The theory for stack is:

```
axioms          isempty(new) = true
                isempty(insert(s,e)) = false
                top(push(new,e)) = e
```

```
                    pop(push(s,e)) = s

boolean             ~(true) = false
                    ~(false) = true
                    (true & b) = b
                    (false & b) = false
                    (true ¦ b) true
                    (false ¦ b) = b
                    (true => b) = b
                    (false => b) = true
                    (true <=> b) = b
                    (false <=> b) = b

equality            (x=x)
                    (x=y) = (y=x)
                    ((x=y) & (y=z)) => (z=x))

induction

reduction           ((Subst(top,1,s2)=Subst(top,1,s2) &
                      Subst(pop,1,s1)=Subst(pop,1,s2) &
                      Subst(isempty,1,s1)=Subst(isempty,1,s2))

                         => (s1=s2))
```

"Higher order" operators on theories or models are avoided, the combining operations (**include**, **import** and **assume**) operate on the text for simplicity. A powerful **with** list is provided for renaming operators or sorts from combined traits. The text is parameterized rather than the theory, thus sidestepping the subtle semantic problems of parameterization.

Semantic Checking

The Larch shared language is designed to enable extensive checking of the specifications as they are constructed. The semantic checks are designed to catch errors expected to be common, and include methods, such as **constrains** and **consequences**, which exist only to provide checkable redundancy. All of the semantic checks rely on a theorem prover.

The semantic checks require that each trait be logically consistent, discharge the assumptions of its external traits, be a conservative extension of its imported traits, be properly constraining, and imply its consequences.

For the stack example these checks would be:

consistency	consistent if the theory does not contain the equation true=false
assumptions	**container** is assumed, but it is also specifically included, so no checking is required
imports	the trait is a conservative extension of the standard imported trait **boolean** if there are no axioms contradicting the meanings of operators found in boolean (eg true & false = true)
constraints	stack is properly constraining if it

	implies properties of the operators in its constrains clause, ie **push**, **pop** and **top**
consequences	**IsEmpty** is the only trait with consequences, any term of the form isempty() must be provably equal to a term not containing isempty (eg isempty(new) and isempty(insert(c,e)))

Detailed discussion of these checks is delayed until the next chapter, where they are presented together with the methods developed for implementing them.

3.1.2 INTERFACE LANGUAGES

An interface separates an implementation of an abstraction from the clients who use the abstraction. Clients depend on the function documented in the interface specification. Defining interfaces is the most important part of system design. Usually it is the most difficult since it must satisfy conflicting constraints of completeness and simplicity and admit a small, fast implementation.

Families of interface languages

A critical part of the interface concerns communication with the environment. Since communication mechanisms differ between programming languages, the interface specification can be more precise, clearer and shorter if they use an interface language which reflects the chosen programming language - hence a family of specification languages.

Theories associated with specifications give meaning to operators appearing in specifications in Larch interface languages. These provide information about actual program modules and are programming language dependent in choice of reserved words and modularization methods, handling of errors and side-effects etc.
The semantics of the interface languages mirrors that of the programming language itself, and can be equally complex. The advantage of this is that we can be precise in what we mean for an implementation to 'satisfy' a specification.

Combining the Algebraic and Predicative languages

The style of specification resembles that used in operational specifications built on abstract models, except that the theory is not a model, and the interface language is built around predicates. Larch follows the notation of predicates on two states, as developed and used by Turing, Hoare, Dijkstra and others.

A specification in the interface language of a data abstraction has three parts:

* a header giving the type name and public routines
* an associated trait and a mapping from the types to the sorts
* interface specifications for each routine which has three parts:
    * a header giving the routine name and its formal parameters
    * an associated trait providing the theory of the operators in its body
    * a body stating any requirements on the routines parameters, and the effects of the routine when the requirements are met.

Example Larch/Pascal

As this report is primarily concerned with Shared Language traits, only one example is given to illustrate how the Shared Langauge and interface languages are combined. Here is a Larch/Pascal specification of a data abstraction with a type, three procedures and a function:

```
type Stk exports stknew, stktop, stkpop, stkpush
     based on sort Stk from Stack with [integer for E]
     procedure stknew(var s : stk)
          modifies at most [ s ]
          ensures s_post = new
     procedure stkpush(var s : stk; e : integer)
          modifies at most [ s ]
          ensures s_post = push(s,e)
     procedure stkpop(var s : stk)
          modifies at most [ s ]
          ensures s_post = pop(s)
     function stktop(s : stk; var e : integer) : boolean
          modifies at most [ e ]
          ensures if ~isempty(s)
               then stktop & e_post = top(s)
               else ~stktop & modifies nothing
```

The trait for Stack was presented before the interface specification, but in practise this need not be the case. Traits need not correspond to a single abstract data type (eg next, rest), whereas interface specifications usually do.

3.2 DESIGN AND SPECIFICATION

Approach

A typical top-down approach to design using Larch would be to develop an understanding of the problem, then write the header information for the interface tier and the syntactic information for the shared language tier (eg operator signatures). The blanks can then be filled in. Next one's understanding of the problem is checked against the formalization and the process repeated as often as necessary.

[Guttag & Horning 86] outline similar steps for reviewing a design. First the design is introduced informally. Then the shared language part is presented and questions about the abstraction formulated. The specification is examined for answers, and the answers evaluated, looking at alternatives. The same process is followed for the interface specifications.

The specification presented here was developed in conjunction with the program - sometimes after program coding (in the case of well understood procedures such as parsing), sometimes during design (in the case of difficulty), otherwise before (once the value of specification had been demonstrated!).

The specification was particularly helpful in testing the parsing routines, an area which is hard to test completely without generating every possible type of specification.

The specification is presented informally with many parts only

partially specified. The partial nature of the specification highlights the need for machine support. Formal specifications are likely to remain a half-completed academic exercise without aids to help keep track of growing specifications (eg library managers and browsers), aids to encourage a reluctant human to throw away a first attempt and start again (eg editors), and means to present alternative viewpoints (paraphrasers) and to reason about the specification (semantic checkers with theorem proving capabilities).

Some of the standard traits in the Larch handbook [Guttag, Horning, Wing 85] have been used.

Design Principles

The parser is not the major research interest of this project, therefore it was kept simple. It uses established parsing techniques and data structures, input and output are text files, and there is no fancy user interface (it is executed from the command line).

Input files

The input to the parser is a simple text file containing one specification. The following specifications taken from the Larch Handbook [Guttag, Horning, Wing 85 pp72-76] are each examples of possible input files:

```
Contain : trait
    introduces
        new : -> C
```

```
        insert : C, E -> C
    asserts C generated by [new, insert]

Isempty : trait
    assumes Contain
    introduces isempty : C -> bool
    asserts for all [ e : E, c : C]
        isempty(new) = true
        isempty(insert(c,e)) = false
    implies converts [isempty]

Next : trait
    assumes Contain
    introduces next : C -> E
    constrains next, insert so that for all [ e : E]
       next(insert(new,e)) = e
    exempts next(new)

Rest : trait
    assumes Contain
    introduces rest : C -> E
    constrains rest so that for all [ e : E]
       rest(insert(new,e)) = new
    exempts rest(new)

Enum : trait
    imports isempty, Next, Rest
    includes Contain
    constrains C so that C partitioned by [ next, rest,

                                            isempty ]
```

Parsing and context sensitive checks

A lexical analyzer reads the input text. The parser uses recursive descent parsing techniques requiring only one pass through the input text.

The parser builds data structures for use by the semantic checker. The symbols (operators, sorts, variables) are all stored in alphabetically sorted binary trees. Expressions are stored in binary trees. Axioms are stored as pairs of expressions. Lists are used for lists of operators or sorts (such as the **generated by** lists).

The parser generates error and warning messages where necessary, and invokes the semantic checker when appropriate. Attention was paid to detail in the error messages, particularly those due to semantic errors, since the checks required are complex. The name of the trait in which the error occured is always given, along with the line and column number and a description of the error. If it is helpful, additional information (such as the expression being evaluated, what it reduces to, what is expected etc) is also provided.

Output

There are two output files. The first contains a list of any errors, both syntactic and semantic. The second file is an aid to understanding and correcting the specification and descibes:

* the relationships between specifications, giving a list of all other specifications assumed, imported or included.

* the elements in the final expanded specification, giving a list of all the sorts, variables, and operators with their domains and range and the computer-generated weight (rank) and status. A list of converted or constrained operators is also provided, as these are subject to special checks.

* the resulting theory, listing all the axioms, and the theorems to be proved (consequences, assumptions, exempts, constrains) as well as the final reduction rules.

This file was found to be extremely useful, and fulfills some of the need for alternative points of view.

The error output file for **Enum** would be

```
LARCH Translator Specification:   enum.lch

enum.LCH (l 4, c 16) - warning (42) May not constrain an
                                imported operator isempty
enum.LCH (l 4, c 16) - warning (42) May not constrain an
                                imported operator next
enum.LCH (l 4, c 16) - warning (42) May not constrain an
                                imported operator rest
0 Errors found
```

The second output file for **Enum** would be

```
LARCH Translator Specification:   enum.LCH

RELATIONSHIPS

Level 0
     Enum Imports  :Isempty, Next, Rest
     Enum Includes :Contain
Level 1
     IsEmpty Assumes  :Contain
     Next Assumes  :Contain
     Rest Assumes  :Contain

ELEMENTS

        Sort
-------------------------------------------------------------------

           C
           E
        bool

Operator Weight      Range     Domains
-------------------------------------------------------------------

      &       5       bool        bool          bool
    <=>       7       bool        bool          bool
     =>       6       bool        bool          bool
  false       2       bool
 insert       9          C           C             E
isempty      10       bool           C
    new       8          C
```

next	11	E	C	
rest	12	E	C	
true	1	bool		
|	4	bool	bool	bool
~	3	bool	bool	

Constrained Operators

new, insert, next, rest, isempty

Variable	Sort	Trait
b	bool	boolean
c	C	isempty
e	E	isempty

THEORY

Axioms

```
 1 :  ~  true =  false
 2 :  ~  false =  true
 3 :  false  &  b =  false
 4 :  true  ¦  b =  true
 5 :  false  ¦  b =  b
 6 :  true  &  b =  b
 7 :  true  =>  b =  b
 8 :  false  <=>  b =  ~  b
 9 :  false  =>  b =  true
10 :  true  <=>  b =  b
11 :  isempty( new ) =  true
12 :  isempty( insert( c , e ) ) =  false
13 :  next( insert( new , e ) ) =  e
14 :  rest( insert( new , e ) ) =  new
```

Exempt

```
 1 :  next( new )
 2 :  rest( new )
```

Reduction rules

```
 1 :  ~  true ->  false
 2 :  ~  false ->  true
 3 :  false  &  b ->  false
 4 :  true  ¦  b ->  true
 5 :  false  ¦  b ->  b
```

```
 6 :  true  &  b ->  b
 7 :  true  =>  b ->  b
 8 :  false  <=>  b ->  false
 9 :  false  =>  b ->  true
10 :  true  <=>  b ->  b
11 :  isempty( new ) ->  true
12 :  isempty( insert( c , e ) ) ->  false
13 :  next( insert( new , e ) ) ->  e
14 :  rest( insert( new , e ) ) -> new
```

Further examples of input and output can be found in Appendix B.

(Note that the Larch Shared language Guide states [Guttag, Horning & Wing 85 p28] "Operators appearing in imported traits may not be constrained by either the importing trait or by any other imported trait". Yet enum constrains all operators with C in their signature, including operators in the imported traits, hence the warning message 42)

3.2.1 TEXT, SYMBOLS AND LEXICAL ANALYZER

Character Sets, Reserved words and Symbols

The specification for the set of legal characters, the delimiter characters and other special characters has been omitted. It is trivial to specify: each character is a constructor with operations such as delimiter shown here:

```
Char : trait
introduces
     a          :          -> char
     space      :          -> char
     delimiter : char      -> bool
asserts for all []
     ~ delimiter(a)
     delimiter(space)
```

Likewise each reserved word or symbol, such as identifier, would be a constructor in the trait **Symbol**. This specification is also trivial and has been ommitted.

The **text** input (or output) is specified as a sequence of Chars which may be added at the end with addlast, or the front with addfirst. Chars are removed using deletelast or deletefirst.

```
Text : trait
includes Char, Symbol
introduces
     newtext    :              -> text
     isnewtext : text          -> bool
```

```
     addlast   : text, char   -> text
     addfirst  : text, char   -> text
     getchar   : text         -> char
     deletelast: text         -> text
     deletefirst:text         -> text
asserts Text generated by [ newtext, addlast, addfirst ]
        Text partitioned by [ isnewtext, getchar, deletelast,
                              deletefirst ]
for all [ t : text, c : char ]
     isnewtext(newtext)
     ~ isnewtext(addlast(t,c))
     ~ isnewtext(addfirst(t,c))
     getchar(addlast(t,c)) = if isnewtext(t) then c
                               else getchar(t)
     getchar(addfirst(t,c)) = c
     deletefirst(addlast(t,c)) = if isnewtext(t) then newtext
                              else addlast(deletefirst(t),c)
     deletefirst(addfirst(t,c)) = t
     deletelast(addlast(t,c)) = t
     deletelast(addfirst(t,c)) = if isnewtext(t) then newtext
                              else addfirst(deletelast(t),c)
exempts
     getchar(newtext),
     deletelast(newtext),
     deletefirst(newtext)
```

The Lexical Analyzer

The trait **Lex** removes a Text sequence with getsymbol, and pushes it back with putsymbol.

```
Lex : trait
assumes Char, Symbol
includes Text
introduces
     getsymbol : text, text   -> text
     putsymbol : text, text   -> text
     symbol    : text, symbol -> bool
asserts
for all [ t : text, c : char, s : text ]
     getsymbol(deletefirst(t,c),s) = if delimiter(c) then s
                    else getsymbol(t,addlast(s,c))
     putsymbol(t,newtext) = t
     putsymbol(t,deletelast(s,c)) = putsymbol(addfirst(t,c),s)

implies exempts
     getsymbol(newtext,s)
```

It is not difficult to extend this simple version of Lex to handle the finer details, such as pushing back delimiters which are part of the next symbol (eg '[',':'), gobbling white space and comments etc. The operation symbol (not given here) has the value true if, for example, the text given is the sequence of char t,r,a,i,t and it is looking for a traitsym.

3.2.2 DATA STRUCTURES

Generic Data Structures

First the generic data structures - pairs, linked lists (FIFO and LIFO), trees and sorted binary trees are specified. The particular data structures, eg Sorts and SortTree, can then be specified in terms of these.

```
Pair : trait
introduces
     <#;#>      : T1, T2  -> C
     #.first    : C       -> T1
     #.second   : C       -> T2
asserts C generated by [ <#;#> ]
        C partitioned by [ .first, .second ]
for all [ f : T1, s : T2 ]
     <f;s>.first = f
     <f;s>.second = s
implies converts [ .first, .second ]
```

In **Flist**, items are inserted at the rear, and removed off the front. In **LList**, items are inserted at the front and removed off the front. **Appendlist** describes joining two lists.

```
FList : trait
introduces
     newlist   :              -> FList
     isnewlist : FList        -> Bool
     insert    : FList, item  -> FList
     front     : FList        -> item
     delete    : FList        -> Flist
asserts FList generated by [ newlist, insert ]
        FList partitioned by [ isnewlist, front ]
for all [ l, r : FList, d : item ]
     isnewlist(newlist)
     ~ isnewlist(insert(l,d))
     front(insert(l,d)) = if isnewlist(l) then d
                          else front(l)
     delete(insert(l,d)) = if isnewlist(l) then l
                         else insert(delete(l),d)
exempts
     front(newlist),
     delete(newlist)
```

```
LList : trait
introduces
     newlist   :              -> LList
     isnewlist : LList        -> Bool
     insert    : LList, item  -> LList
     front     : LList        -> item
     appendlist: LList, LList -> LList
asserts LList generated by [ newlist, insert ]
        LList partitioned by [ isnewlist, front ]
for all [ l, r : LList, d : item ]
     isnewlist(newlist)
     ~ isnewlist(insert(l,d))
     front(insert(l,d)) = d
     appendlist(l,newlist) = l
     appendlist(l,insert(r,d)) = insert(appendlist(l,r),d)
exempts
     front(newlist)
```

BSTree is a binary sorted tree. **Make** is an example of a "hidden" or auxillary operation, that is needed to ensure the tree is sorted, but would not be visible to the user in the Interface specification.

```
BSTree : trait
introduces
     newtree   :                          -> BSTree
     isnewtree : BSTree                   -> Bool
     make      : BSTree, item, BSTree     -> BSTree
     insert    : BSTree, item             -> BSTree
     left      : BSTree                   -> BSTree
     right     : BSTree                   -> BSTree
     find      : BSTree, item             -> Bool
     content   : BSTree                   -> BSTree
asserts
     BSTree generated by [ newtree, make ]
     BSTree partitioned by [ left, right, find, content ]
for all [ l, r : BSTree, d, e : item ]
     isnewtree(newtree)
     ~ isnewtree(make(l,d,r))
     insert(newtree,e) = make(newtree,e,newtree)
     insert(make(l,d,r),e) =
          if d = e then make(l, d, r)
          else if d < e then make(l,d,insert(r,e))
               else make(insert(l,e),d,r)
     left(newtree) = newtree
     left(make(l,d,r)) = l
     right(newtree) = newtree
     right(make(l,d,r)) = r
     find(newtree,d) = false
```

```
    find(make(l,d,r),e) = if d = e then true
                          else if d < e then find(r,e)
                               else find(l,e)
    content(make(l,d,r)) = d
converts [ left, right, find, content ]
exempts
    content(newtree)
```

Sorts, Operators and Variables

The sorts, operators and variables are stored in sorted binary trees. **Sorts** are unstructured text, **variables** are pairs (and could have been specified using Pair), and **operators** are more complex, incorporating **Sorts** and the Handbook trait **Cardinal**.

```
Sort : trait
includes Text with [ sortid for text ]

SortTree : trait
includes Sort,
         BSTree with [ text for item, entersort for insert,
                  findsort for find, SortTree for BSTree ]

SortList : trait
includes Sort,
         FList with [ text for item, SortList for Flist ]


Variable : trait
includes Sort, Text with [ varid for text ]
introduces
     (- #;# -) : varid, sortid-> variable
     #.varid   : varid        -> text
     #.varsort : sortid       -> text
asserts Variable generated by [ (- #;# -) ]
     Variable partitioned by [ .varid, .varsort ]
for all [ i : varid, s : sortid ]
     (- i;s -).varid = i
     (- i;s -).varsort = s
```

```
VariableTree : trait
includes Variable,
         BSTree with [(- #;# -) for item, entervar for insert,
                 findvar for find, VariableTree for BSTree]

Operator : trait
includes Cardinal, Sort, SortList, Text with [ opid for text ]
introduces
     (* #;#;#,# *): opid, cardinal, sortlist, sortid ->
operator
     #.opid       : operator            -> opid
     #.opweight   : operator            -> cardinal
     #.opdomains  : operator            -> Sortlist
     #.oprange    : operator            -> sortid
asserts Operator generated by [ (* #;#;#;# *) ]
     Operator partitioned by [ #.opid, #.opweight,#.opdomains,
                               #.oprange ]
for all [ i : opid, w : cardinal, d : SortList, r : sortid ]
     (* i;w;d;r *).opid = i
     (* i;w;d;r *).opweight = w
     (* i;w;d;r *).opdomains = d
     (* i;w;d;r *).oprange = r
OperatorTree : trait
includes Operator,
         BSTree with [ (* #;#;#;# *) for item,
                       enterop for insert, findop for find,
                       Operatortree for BSTree ]

OperatorList : trait
includes Operator,
     FList with [(* #;#;#;# *) for item,
                 OperatorList for Flist]
```

Expressions

Unfortunately there is no mechanism for variable sorts (corresponding to "unions" in C) as is needed for an expression which could be a variable or an operator. As a result, the **exprid** in **Expression** is unstructured text to accomodate both.

```
Expression : trait
includes Text with [ exprid for text ]

ExpressionTree : trait
includes Expression
introduces
     newexpr  :                              -> ExpressionTree
     enterleaf: Expression                   -> ExpressionTree
     enterexpr: ExpressionTree, Expression, ExpressionTree
                                             -> ExpressionTree
     left     : ExpressionTree               -> ExpressionTree
     right    : ExpressionTree               -> ExpressionTree
     isleaf   : ExpressionTree               -> bool
     content  : ExpressionTree               -> Expression
asserts
     ExpressionTree generated by
           [ newexpr,enterleaf,enterexpr]
     Expressiontree partitioned by
          [left, right, isleaf, content]
for all [ l, r : ExpressionTree, d : expression ]
     isleaf(newexpr) = false
     isleaf(enterleaf(d))
     ~ isleaf(enterexpr(l,d,r))
     left(enterexpr(l,d,r)) = l
```

```
      right(enterexpr(l,d,r)) = r
      content(enterleaf(d)) = d
      content(enterexpr(l,d,r)) = d
implies converts [ newexpr, isleaf, left, right, content ]
exempts
      left(enterleaf(d))
      left(newexpr)
      right(enterleaf(d))
      right(newexpr)
      content(newexpr)
```

Axioms

Axioms are pairs, comprising two expressions. The axioms are stored in a list.

```
Axiom : trait
includes ExpressionTree,
     Pair with [ {! #;# !} for < #;# >,
                 Axiom for C,
                 ExpressionTree for T1, ExpressionTree for T2,
                 #.lhs for #.first, #.rhs for #.second ]

AxiomList : trait
includes Axiom,
     FList with [ {! #;# !} for item, AxiomList for Flist]
```

3.2.3 PARSER

Introduction

The specification given here is for the **kernel language**, but can be easily extended.

The basic idea is for each non-terminal in the grammar of the form:

non-terminal1 ::= **terminal1** non-terminal2

include axioms:

```
non-terminal1(newtext) = false
non-terminal1(getsymbol(t,s)) = if symbol(s,terminal1sym)
                                then non-terminal2(t)
                                else false
```

The operation putsymbol, which reverses the effect of getsymbol, is used to reduce the depth of if-then-elses and make the specification easier to read.

It is interesting to compare the structure, length and readability of the parser specification with the Backaus-Naur specification for the Larch grammar given in Appendix A.

Errors

The specification for parsing appears to neglect error recovery, this could be dealt with at the level of the interface specification.

```
Parse : trait
includes
     Symbol, Text, Lex
introduces
%    all operators are of the form
%         operator       : text    -> bool
%    and have been ommitted
asserts for all [ t, s : text ]

     parse(newtext)
     parse(getsymbol(t,s)) = if symbol(s,identifier)
                    then traitcolon(t) else false
     traitcolon(newtext) = false
     traitcolon(getsymbol(t,s)) = if symbol(s,colon)
                    then trait(t) else false
     trait(newtext) = false
     trait(getsymbol(t,s)) = if symbol(s,traitsym)
                    then traitbody(t) else false
     traitbody(newtext)
     traitbody(getsymbol(t,s)) = if symbol(s,introducesym)
                    then oppart(t) else
proppart(putsymbol(t,s))
%
%    OPPART
%
     oppart(newtext)
     oppart(getsymbol(t,s)) = if symbol(s,identifier)
                    then opcolon(t) else
opform(putsymbol(t,s))
     opcolon(newtext) = false
     opcolon(getsymbol(t,s)) = if symbol(s,colon)
                    then signature(t) else false
```

```
     signature(newtext) = false
     signature(getsymbol(t,s)) = if symbol(s,identifier)
                    then sortcomma(t) else
arrow(putsymbol(t,s))
     sortcomma(newtext) = false
     sortcomma(getsymbol(t,s)) = if symbol(s,comma)
                    then sortlist(t) else
arrow(putsymbol(t,s))
     sortlist(newtext) = false
     sortlist(getsymbol(t,s)) = if symbol(s,identifier)
                    then sortcomma(t) else false
     arrow(newtext) = false
     arrow(getsymbol(t,s)) = if symbol(s,arrowsym)
                    then range(t) else false
     range(newtext) = false
     range(getsymbol(t,s)) = if symbol(s,identifier)
                    then oppart(t) else false
     opform(newtext)
     opform(getsymbol(t,s)) = if symbol(s,hash)
                    then firstopsym(t)
                    else if symbol(s,opsym) then hashopsym(t)
                    else proppart(putsymbol(t,s))
     firstopsym(newtext) = false
     firstopsym(getsymbol(t,s)) = if symbol(s,identifier)
                    then hashopsym(t) else false
     hashopsym(newtext) = false
     hashopsym((getsymbol(t,s)) = if symbol(s,hash)
                    then nextopsym(t)
                    else opcolon(putsymbol(t,s))
     nextopsym(newtext) = false
     nextopsym(getsymbol(t,s)) = if symbol(s,opsym)
                    then hashopsym(t)
```

```
                    else opcolon(putsymbol(t,s))
%
%     PROPPART
%
      proppart(newtext)
      proppart(getsymbol(t,s)) = if symbol(s,assertsym)
                    then props(t) else false
      props(newtext)
      props(getsymbol(t,s)) = if symbol(s,identifer)
                    then generatedby(t)
                    else propsaxioms(putsymbol(t,s))
%     GENERATORS
      generatedby(newtext) = false
      generatedby(getsymbol(t,s)) = is symbol(s,generated)
                    then genbylist(t)
                    else partitionedby(putsymbol(t,s))
      genbylist(newtext)
      genbylist(getsymbol(t,s)) = if symbol(s,bysym)
                    then genopensq(t)
                    else props(putsymbol(t,s))
      genopensq(newtext) = false
      genopensq(getsymbol(t,s)) = if symbol(s,opensq)
                    then genoplist(t) else false
      genoplist(newtext) = false
      genoplist(getsymbol(t,s)) = if symbol(s,identifier)
                    then genopcomma(t)
                    else genclosesq(putsymbol(t,s))
      genopcomma(newtext) = false
      genopcomma(getsymbol(t,s)) = if symbol(s,comma)
                    then genoplistmore(t)
                    else genclosesq(putsymbol(t,s))
      genoplistmore(newtext) = false
```

```
genoplistmore(getsymbol(t,s)) = if symbol(s,identifier)
                then genopcomma(t) else false
genclosesq(newtext) = false
genclosesq(getsymbol(t,s)) = if symbol(s,closesq)
                then genbycomma(t) else false
genbycomma(newtext)
genbycomma(getsymbol(t,s)) = if symbol(s,comma)
                then genbymore(t)
                else partitions(putsymbol(t,s))
genbymore(newtext) = false
genbymore(getsymbol(t,s)) = if symbol(s,identifier)
                then genbylist(t) else false
%   PARTITIONS - Omitted, similar to Generators above
%   AXIOMS - VARIABLE DECLARATIONS
propsaxioms(newtext)
propsaxioms(getsymbol(t,s)) = if symbol(s,forsym)
                then all(t) else false
all(newtext) = false
all(getsymbol(t,s)) = if symbol(s,allsym)
                then varopensq(t) else false
varopensq(newtext) = false
varopensq(getsymbol(t,s)) = if symbol(s,opensqsym)
                then vardcl(t) else false
vardcl(newtext) = false
vardcl(getsymbol(t,s)) = if symbol(s,identifier)
                then varidcomma(t)
                else varcolon(putsymbol(t,s))
varidcomma(newtext) = false
varidcomma(getsymbol(t,s)) = if symbol(s,comma)
                then varidlist(t)
                else varcolon(putsymbol(t,s))
varidlist(newtext) = false
```

```
varidlist(getsymbol(t,s)) = if symbol(s,identifier)
                then varidcomma(t) else false
varcolon(newtext) = false
varcolon(getsymbol(t,s)) = if symbol(s,colon)
                then varsortid(t)
                else varclosesq(putsymbol(t,s))
varsortid(newtext) = false
varsortid(getsymbol(t,s)) = if symbol(s,identifier)
                then vardclcomma(t) else false
vardclcomma(newtext) = false
vardclcomma(getsymbol(t,s)) = if symbol(s,comma)
                then vardcllist(t)
                else varclosesq(putsymbol(t,s))
vardcllist(newtext) = false
vardcllist(getsymbol(t,s)) = if symbol(s,identifier)
                then varidcomma(t) else false
varclosesq(newtext) = false
varclosesq(getsymbol(t,s)) = if symbol(s,closesqsym)
                then equations(t) else false
%    AXIOMS - EQUATIONS
equations(newtext)
equations(getsymbol(t,s)) = if symbol(s,identifier)
                then term1(t)
                else propsaxioms(putsymbol(t,s))
term1(newtext)
term1(getsymbol(t,s)) = if symbol(s,leftcurly)
                then termlist(t)
                else equals(putsymbol(t,s))
equals(newtext) = false
equals(getsymbol(t,s)) = if symbol(s,equalsym)
                then term2(t) else false
```

```
%    Termlist is a list of terms separated by commas and can
be % specified in a similar manner to the other lists (eg
%    generators).  Term2, the right hand term, is similar to
%    term1.
%    All operators are convertible
```

Example

A trait specification

```
     id : trait
```

would be expressed

```
     parse(
      getsymbol(
          getsymbol(
               getsymbol(newtext,traitsym),colon),identifier))
```

and would be reduced by the following stages

```
     traitcolon(
        getsymbol(
          getsymbol(newtext,traitsym),colon))

     trait(getsymbol(newtext,traitsym))

     traitbody(newtext)

     true
```

Translating Extensions to the Kernel Language

The kernel language is extended to include consequences, mixfix operators, implicit signatures and partial opforms etc. Parse can be easily extended to include these.

As an example, the partial specification below shows how external references (assumes, includes and imports) could be handled:

```
ExtendedParse : trait
includes Parse
introduces
     fetch   : text      -> text
     assumes : text      -> Bool
asserts
for all [ t, s : text ]
     assumes(newtext)
     assumes(getsymbol(t,s)) =
          if symbol(s,identifier) then
               (if parse(fetch(s)) then
                    assumecomma(t)
               else false)
          else externals1(putsymbol(t,s))
```

Fetch is an operation which fetches the text of the external trait. Parse can be extended to enter the name of the referenced trait in a RefTrait tree, Rtree, and recursive trait references checked for by changing line 11 to

```
               if ~ findtrait(rtree,s) & parse(fetch(s)) ...
```

3.2.4 CONTEXT-SENSITIVE CHECKING

Simple Traits

* The set of sortids, opids and varids must be disjoint.
* Each sortid and each sortedop appearing anywhere in the trait must appear in its oppart.
* Each varid in an axiom must appear in exactly one vardcl.
* No varid may occur more than once in vardcl.

The trait now includes the specifications Operator, Operatortree, Sort, Sorttree, Variable and Variabletree. The axioms parsing opdcl and vardcl can be extended as shown for oppart below:

```
asserts for all [Stree:SortTree, OTree:OperatorTree, t,s:Text]
     oppart(newtext)
     oppart(getsymbol(t,s)) =
          if symbol(s,identifer) then
            (if findsort(Stree,s) then false
             else
             findop(enterop(Otree,{* s,0,newlist,newtext*}))
                & opcolon(t))
          else proppart(putsymbol(t,s))
```

In the specification above, findop(enterop(Otree,s)) is always true, and is an inelegant way of ensuring the operator is entered into the tree. This raises the question of operators "introduced" more than once - no mention is made in the Larch documentation of this.

Generators and Partitions

* The range of each sortedop in a generators must be the Sortid of the generators.
* The domain of each sortedop in a partitions must include the sortId of the partitions.
* At least one sortedop in each generated bylist must have a domain in which the sortid of the generators does not occur.
* The range of at least one sortedop in each partitioned bylist must be different from the sortid of the partitions.

A Generator and Partition data structure, which is a pair of SortId and OpList, is required. These then become items in GeneratorList and PartitionList (using the specification FList). Including these in the trait, the above checks can be accomplished in a fairly straightforward manner.

All of the other context-sensitive checks required can be specified using the data structures outlined and the operations on them.

3.2.5 IMPLEMENTATION DETAILS

Machine and Language

The system is implemented in TURBO C on an IBM-compatible PC using MS-DOS.

Files

Input files are all named **TRAIT.LCH**, where TRAIT is the traitid. This is necessary in order to identify the text file for a trait being assumed, imported or included. The output files generated are named **TRAIT.ERR** and **TRAIT.DMP** respectively.

Data Structures

The data structures are binary sorted trees, binary trees, and lists, and established algorithms for implementing them (entering data, searching, copying, deleting etc) were used.

The expression trees were slightly modified to distinguish between mixfix operators, eg

```
if # then # else
```

as well as functional operators, eg

```
top(insert(c,e)).
```

This was neccessary to output expressions as the user had input them.

In detail the data structures for a trait are:

* Operators, Sorts and Variables (trees)
* Operators and Constrained Operators (lists of operators)
* Axioms, Rules, Exempts, Implies, Assumes, Imports, Constrains (lists of expression pairs)
* Converts, Generators, Partitions (lists of a sort and associated operators)
* Assumed, Imported and Included traits (lists of names for recursive checks)

The data structures were refined to facilitate easy implementation of the semantic checks. If necessary, redundant data was stored if it made checking simpler, for example both a sorted operator tree and a list of operators as they were entered was maintained.

The data structure is stored entirely in memory which may pose limitations on the size of specification that can be processed.

CHAPTER 4 : THE SEMANTIC CHECKER

4.1 THEOREM PROVING

The implementation of the Larch Shared language requires a theorem prover for use by the semantic checker. The semantic checks required were presented briefly in Chapter 3. They are more fully described along with a description of how they were implemented using the theorem prover in the next section of this chapter.

Prolog was initially investigated as a suitable language for implementation of the Shared Language since it is well suited to writing parsers [Warren 80] [Cohen 85] and has built-in theorem proving by unification and resolution [Genesereth & Ginsberg 85]. Prolog programs are similar to Larch specifications and methods were developed for translating Larch into Prolog. However, Prolog was found to be unsuitable for various reasons. For further detail and examples, see Appendix C.

Term Rewriting System methods were ultimately chosen to implement the theorem prover, mainly on the recommendation in [Guttag, Horning and Wing 85 p42].

4.1.1 SUBSTITUTION AND UNIFICATION

Definitions

A substitution is a finite set of ordered pairs of the form $\{(t_1,v_1),\ldots,(t_n,v_n)\}$ where every v_i is a variable and t_i is a term not containing v_i, and no two pairs in the set have the same variable.

If δ is a substitution $\{(t_1,v_1),\ldots,(t_n,v_n)\}$ and E is an expression, $E\delta$ is an expression obtained by simultaneously replacing each occurrence of the variable v_i by the term t_i, i=1..n.

The composition of substitutions is associative.

A substitution δ is called a unifier for a set $W=\{E_1,\ldots,E_n\}$ if, and only if, $E_1\delta = E_2\delta = \ldots = E_n\delta$. The set W is said to be unifiable if there is a unifier for it. A unifier for a set $W=\{E_1,..,E_n\}$ is said to be the most general unifier (mgu) if, and only if, for each unifier for the set there is a substitution μ such that $=\delta\mu$.

The Disagreement set D of a set W of expressions is obtained by locating the first symbol, reading left to right, at which the expressions differ, and then extracting from each expression in W the subterm which has that symbol as its root.

The Unification Algorithm

Step 1: Set k=0, W_k = W, and δ_k = { }

Step 2: If W_k is a singleton, stop - δ_k is the most general unifier. Otherwise find the disagreement set D_k.

Step 3: If there exists elements v_k and t_k in D_k, such that vk is a variable that does not occur in term t_k, goto Step 4. Otherwise stop - W is not unifiable.

Step 4: Let $\delta_{k+1} = \delta_k\{(t_k,v_k)\}$ and $W_{k+1} = W_k\{(t_k,v_k)\}$.

Step 5: k = k+1. Goto Step 2.

By Robinson's Unification Theorem, this algorithm will always terminate given W, a finite set of expressions, and the last δ_k is the most general unifier for W [Chang & Lee 73].

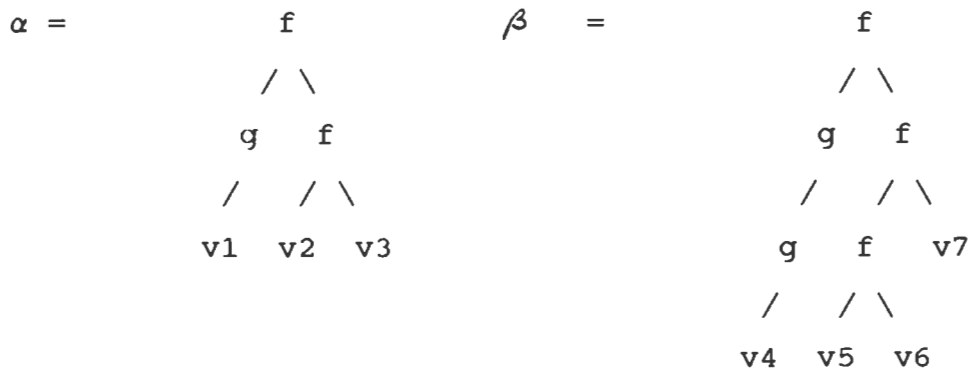
Example

Given two expressions

α = f(g(v1),f(v2,v3))

β = f(g(g(v4)),f(f(v5,v6),v7))

or in tree form



Starting with

$k=0$,

$W_0=\{f(g(v1),f(v2,v3)),\ f(g(g(v4)),f(f(v5,v6),v7))\}$

$\delta_0=\{\}$,

$D_0=\{v1,(g(v4))\}$,

substitute g(v4) for all instances of v1 and obtain

$k=1$,

$W^k=\{f(g(g(v4)),f(v2,v3),\ f(g(g(v4)),f(f(f(v5,v6),v7))\}$

$\delta_1=\{(g(v4),v1)\}$,

$D_1=\{(f(v4,v5),v2)\}$,

substitute f(v4,v5) for all instances of v2 and obtain

$k=2$,

$W_2=\{f(g(g(v4)),f(f(v4,v5),v3)),$

$f(g(g(v4)),f(f(v4,v5),v6))\}$,

$\delta_2=\{(f(v4,v5),v2),\ (g(v4),v1)\}$,

$D_2=\{(v3,v6)\}$,

subsitute v3 for all instances of v6 and obtain

$k=3$,

$W_3=\{f(g(g(v4)),f(f(v4,v5),v3)),$

$f(g(g(v4)),f(f(v4,v5),v3)\}$,

$\delta_3=\{(v3,v6),\ (f(v4,v5),v2),\ (g(v4),v1)\}$,

$D_3=\{\}$,

W is unifiable, with δ_3 the most general unifier.

4.1.2 TERM REWRITING SYSTEMS

Larch specifications consist of a set of universally quantified equations describing the operations on a type and their relations to each other. Both the axioms and theorems under consideration are well-formed equalities between expressions. The theorems to be proved can be divided into two categories :

* equational theorems proved by replacing expressions by equal expressions with respect to the equations. These theorems are provable if, and only if, they are valid equalities.
* inductive theorems requiring an inductive rule in addition to equational reasoning.

The Word Problem

The fundamental decision problem for equational theories is the word problem: finding a decision procedure for proving or disproving identities from a set of equations [Knuth & Bendix 83]. Although the word problem is generally unsolvable, many of the word problems for abstract algebras have been shown to be solvable theoretically and practically using a uniform methodology based on term rewriting methods [Lescanne 83 p99].

Term Rewriting Systems

Term rewriting systems are a model of computation used to model formula manipulations in various applications, such as

program optimization, algebraic simplifiers and automatic theorem proving.

A term rewriting system is a triple < F, V, R > where

* F is a set of ranked operators, each f in F has an arity or degree, the number of arguments f acts upon, if the arity is 0, f is a constant.
* V is a set of variables. F and V are disjoint sets.
* R is a set of oriented reduction (rewrite) rules always used from left to right. A reduction is a pair (t,s) written t -> s where t, s are terms built from F and V.

[Bergstra & Klop 86 p326]

The method used to prove an equational theorem $\alpha=\beta$ is to reduce α and β using the reduction rules until one arrives at irreducible terms α^* and β^*. If α^* equals β^* then $\alpha=\beta$ is valid. For this method to be a decision procedure, two problems must be addressed:

* is the irreducible expression α^* associated with α unique?
* does the process of rewriting the expression α always terminate?

Definitions

Well-formed formulas of operators and variables are called terms (words). Subterms of a term α are the term itself, and the subterms $\alpha_1,\ldots,\alpha_m$ if α has the form $f(\alpha_1,\ldots,\alpha_m)$. Nontrivial subterms contain at least one operator symbol which is not a constant. The result of replacing a subterm α_i in term α by a term β, is written $\alpha[\alpha_i <- \beta]$. A term β has the form of α if β can be obtained by substitution from α, that is

β and α are unifiable.

A term α is reducible, with respect to R, if there is a rule t -> s, a substitution δ and a subterm of α, α_1, such that $t\delta = \alpha_1$. The term α can then be rewritten $\alpha[\alpha_1 <- s\delta]$. A term α is irreducible if, with respect to R, there is no α' such that $\alpha -> \alpha'$.

Derivation is a sequence of rewriting steps t -> t1 -> t2 -> ... The notation t ->* tn indicates a derivation sequence t->...->tn. The choice of which rule to apply, and the choice of which subterm to apply a rule to, is made non-deterministically. A term $\beta = R(\alpha)$ is the normal form, with respect to R, of the term α iff $\alpha->*\beta$ and β is irreducible.

A term rewriting system is uniformly terminating if there is no infinite derivation. It is confluent if for all terms, s,u,v such that s->*v, and s->*u, there is a term t such that u->*t and v->*t holds. It is canonical (convergent) if it is terminating and confluent. A convergent term rewriting system determines a decision procedure for the associated equational theory. Both properties, confluence and termination, are undecidable [Huet 80].

4.1.3 UNIFORM TERMINATION

The principal restriction in solving $\alpha = \beta$ with respect to a set of reductions R, is that one must find a strict partial order > such that for each reduction, t -> s, t > s.

The motivation for the definition of > lies in the theorem which shows that if $\alpha > \beta$, and δ is any substitution {t1,t2,...tn}, then $\alpha\delta > \beta\delta$ [Knuth & Bendix 83].

Proving uniform termination

The uniform termination problem for term rewriting systems is an undecidable problem, but algorithms exist which work in most practical situations with little or no intervention from the user. These can be compared and their relative power analyzed.

The methods are based on simplification orderings. A partial ordering on terms is a simplification ordering on terms if two properties hold for all α,β

* Subterm Property:

  $\alpha < f(...,\alpha,...)$

* Compatibility Property:

  $\alpha < \beta \Rightarrow f(...,\alpha,...) < f(...,\beta,...)$.

[Pettorossi 81 p436]

All the methods given here are an extension of a precedence that is an ordering on basic symbols - i.e. operators are ranked.

Method 1: The Knuth-Bendix > relation

[Knuth & Bendix 83] show that the set of all terms is well ordered by the relation >.

Definition

A pure term is one containing no variables, and it's weight W can be calculated by

$$W(\alpha) = \sum_{j \geq 1} W_j \, N(f_j, \alpha)$$

where $N(f_j, \alpha)$ is the number of occurrences of the operator f_j in α, and W_j is the weight of the operator f_j. Operators are ranked by their weights with each operator having positive weight.

The definition of weight W can be extended to included variables

$$W(\alpha) = W_0 \sum_{j \geq 1} N(v_j, \alpha) + \sum_{j \geq 1} W_j N(f_j, \alpha)$$

where W_0 is the minimum weight of a pure term, that is the weight of the nullary operator, $N(v_j, \alpha)$ is the number of occurrences of variable v_j in α.

We can say $\alpha > \beta$ if and only if either:

1 $W(\alpha) > W(\beta)$ and $N(v_i,\alpha) \geq N(v_i,\beta)$ for all $i \geq 1$

or

2 $W(\alpha) = W(\beta)$ and $N(v_i,\alpha) = N(v_i,\beta)$ for all $i \geq 1$ and either

21 α has the form $f_N v_1 .. f_N v_m$ where f_N is a special unary operator of rank 0 and $\alpha = f_N^t v_k$, $\beta = v_k$ for some $t \geq 1$

or

22 $\alpha = f(\alpha_1,\ldots,\alpha_m)$, $\beta = g(\beta_1,\ldots,\beta_n)$ and either

22a $m > n$

or

22b $m = n$ and $\alpha_1 = \beta_1, \ldots, \alpha_{t-1} = \beta_{t-1}$, $\alpha_t > \beta_t$ for some $1 \leq t \leq m$.

Operators in a Larch specification are assigned weights in this implementation as follows: first the constructors in the order they are presented in the generated by list, then the remaining operators as they are presented in the oppart following introduces. It is then easy to design an algorithm to determine if $\alpha > \beta$, or $\beta > \alpha$, or if α and β are unrelated (written $\alpha \# \beta$).

Examples of the > relation

Operators	Weights	
e	1	W_0
*	2	

1) α : x * e β : x

$\Sigma\ N(v_j,\alpha)$: 1 : 1

$\Sigma\ W_jN(f_j,\alpha)$: 1 + 2 : 0

$W(\alpha)$: 4 : 1

Therefore $\alpha > \beta$ since $W(\alpha) > W(\beta)$.

2) α : (x * y) * z β : x * (y * z)

$\Sigma\ N(v_j,\alpha)$: 3 : 3

$\Sigma\ W_jN(f_j,\alpha)$: 2*2 = 4 : 2*2 = 4

$W(\alpha)$: 7 : 7

Since $W(\alpha) = W(\beta)$, we must examine case 2 of the > relation further, in particular case 22b since $\alpha = f(\alpha_1,\alpha_2)$ and $\beta = f(\beta_1,\beta_2)$ where f is *.

For t = 1 to 2 calculate $W(\alpha_t)$ and $W(\beta_t)$.

Since $W(\alpha_1) > W(\beta_1)$, we have $W(\alpha) > W(\beta)$.

Operators	Weights	
f	1	W_0
g	2	

3) α : f(y,y,y,y,z) β : g(z,z,y)

$\Sigma\ N(v_j,\alpha)$: 5 : 3

$\Sigma\ W_jN(f_j,\alpha)$: 1*1 = 2 : 1*2 = 2

$W(\alpha)$: 7 : 5

Now $W(\alpha) > W(\beta)$, and $N(v_1,\alpha) > N(v_1,\beta)$, but $N(v_2,\alpha) < N(v_2,\beta)$ therefore $\alpha \# \beta$.

Limitations

The status of operators is assumed to be left to right. The method fails for commutative operators. There is no way of deciding in general how to construe an axiom such as a+b=b+a as a reduction.

Method 2: Recursive Path Ordering

In addition to a precedence ordering like the Knuth-Bendix ordering, operator symbols have status which can be left-to-right, right-to-left or none. If the operator * is left-to-right, then (x*y)*z > x*(y*z). If the status were right-to-left, x*(y*z) > (x*y)*z. With no status, (x*y)*z and x*(y*z) cannot be ordered, but (x*y)*z > (x*y).

The recursive path ordering (RPO) [Kapur, Narendran, Sivakumar 85] [Pettorossi 81] compares terms, by first comparing the root symbol, then recursively comparing the subterms according to the result of the root comparison.

Definition

A $\alpha=f(\alpha_1,..,\alpha_m) > \beta=x$

iff x is in Var(α), the set of variables in α

B $\alpha=f(\alpha_1,..,\alpha_m) > \beta=g(\beta_1,..,\beta_n)$

iff one of the following three hold:

1 f .> g and $\alpha > \beta_i$ for all i, $1 \leq i \leq n$

2 f ~ g and

21 if f and g have left-right status then

there exists j such that $\alpha_1 \sim \beta_1$,

$\alpha_{j-1} \sim \beta_{j-1}$, $\alpha_j > \beta_j$ and $\alpha > \beta_i$, for $j+1 \leq i \leq n$

Similarly for right-left status $\alpha_n \sim \beta_n$,

$\alpha_{j+1} \sim \beta_{j+1}$, $\alpha_j > \beta_j$ and $\alpha > \beta_i$, for $j-1 \leq i \leq 1$

whereas

22 if f and g have no status then

$\{\alpha_1,..,\alpha_m\} >> \{\beta_1,..,\beta_n\}$

3 $\alpha_i > \beta$ or $\alpha_i \sim \beta$ for some i, $1 \leq i \leq n$

where

1 > is the recursive path ordering

2 .>is the ordering on function symbols

3 ~ is the equivalence relation

$\alpha=f(\alpha_1,..,\alpha_m) \sim \beta=g(\beta_1,..,\beta_n)$ iff

f ~ g, m=n, and there is a permutation p of the set

$\{1,..,n\}$ such that $\alpha_i \sim \beta_{p(i)}$ for all $1 \leq i \leq n$

Equivalent operators should have the same status

4 >> is the multiset extension of the ordering >,

$M_1 >> M_2$ iff for each x in $M_2 - M_1$, there is a y in $M_1 - M_2$ such that $y > x$.

Once again, it is necessary to know which operators are constructors to form a basis for ordering operators. The

recursive path ordering was also implemented, and is preferred to the Knuth-Bendix method as being more powerful. The Larch language has no mechanism for specifying status of operators, so no status is assumed, or left-right if that fails.

Examples

1) The operators need not be ranked.

α : x * e $\qquad\qquad$ β : x

From A, $\alpha > \beta$ since x is a variable in α.

2) The operator * has left-right status.

α : (x * y) * z $\qquad\qquad$ β : x * (y * z)

Since the top level function is the same, by B21 we find $\alpha_j > \beta_j$, and $\alpha > \beta_{j+1}$ with j=1.

(Note, without left-right status the terms are unrelated)

3) The operators are ranked f .> g

α : f(y,y,y,y,z) $\qquad\qquad$ β : g(z,z,y)

From 21 we must show that $\alpha > \beta_i$ for $1 \leq i \leq 3$.

f(y,y,y,y,z) > z, by A. Likewise f(y,y,y,y,z) > y by A.

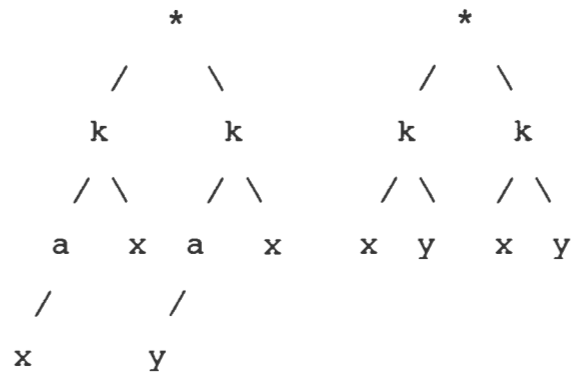
Therefore $\alpha > \beta$.

(Note, this could not be ordered by the Knuth-Bendix method)

Limitations

There are terms which RPO cannot order, but which can be ordered by an intuitive concept of paths: for example the equation a(b(x)) = c(d(c)) with a>d, b>c, is incomparable. Yet one feels a(b(x)) > c(d(x)) since a "takes care of" d and b "takes care of" c. Hence methods such as the ordering of [Kapur et al 85] are based on paths.

Another example of terms which cannot be ordered by RPO, but whose ordering appears obvious is:



Method 3: Path Ordering

The path ordering [Kapur et al 85] attempts to define the "taking care of" notion. The method incorporates operator status, contains RPO and eliminates many of its drawbacks.

Definitions

Var(α) is the set of variables in the term α. The size of a term, α, is the number of variable and function occurrences, denoted $|\alpha|$.

A Path is a sequence of tuples ending possibly in a variable.
Let P = $<f_1,\alpha_1>..<f_n,\alpha_n>\{x\}$ be a path. Then

1 f_i is the top level function symbol of α_i, $1 \leq i \leq n$

2 α_{i+1} is an immediate subterm of α_i, $1 \leq i \leq n$

3 if P ends in x then x is a subterm of α_n

A path is a full path of a term α iff

1 α = x, a variable, then x is the only full path in α

2 α = b, a constant, then <b, b> is the only full path

3 $\alpha = f(t_1,..t_n)$, then a full path is $<f,\alpha>.p$
where p is a full path in some α_i.

For example, if α=f(x,y) then <f, f(x,y)>x and <f, f(x,y)>y are the full paths.

To compare paths ($>_P$):

1 paths ending in different variables are incomparable

2 a variable is incomparable with a tuple (this ensures constants are always greater than variables)

3 paths ending in the same variable are compared by dropping the variable and comparing the remainder of the path

Each tuple in a path can have associated with it a left-context (LC) visualized as above the corresponding node in a tree representation of the term, and a right-context (RC) visualized as below the node.

Let $P_1 = <f_1,\alpha_1>..<f_m,\alpha_m>$ and $P_2 = <g_1,\beta_1>..<g_n,\beta_n>$. $P_1 > P_2$ iff for all $<g_j,\beta_j>$ in P_2 there exists $<f_i,\alpha_i>$ in P1 such that

P1 $f_i > g_j$ or

P2 $f_i \sim g_j$ and if they have no status then

P21 $RC(<f_i,\alpha_i>,P_1) >_P RC(<g_j,\beta_j>,P_2)$ or

P22 $RC(<f_i,\alpha_i>,P_1) \sim RC(<g_j,\beta_j>,P_2)$ and $\alpha_i >_T \beta_j$, or

P23 $RC(<f_i,\alpha_i>,P_1) \sim RC(<g_j,\beta_j>,P_2)$ and $\alpha_i \sim \beta_j$ and $LC(<f_i,\alpha_i>,P_1) >_P LC(<g_j,\beta_j>,P_2)$

If they have left-right status then

P21 $\alpha_i >_T \beta_j$, or

P22 $\alpha_i \sim \beta_j$ and $LC(<f_i,\alpha_i>,P_1) >_P LC(<g_j,\beta_j>,P_2)$

right-left status is similar.

where

1 paths $P_1 \sim P_2$ if m=n and $f_i \sim g_i$ and $\alpha_i \sim \beta_i$, $1 \leq i \leq n$

2 term comparison, $>_T$ is defined next.

Let $M=\{\alpha_1,\alpha_2,..,\alpha_m\}$ be a multiset of terms. Then MP(M) is the union of all full paths in each αj.

To <u>compare terms</u> ($>_T$):

T1 If α is a non-variable term and β is a variable then $\alpha >_T \beta$

iff α contains β

T2 $\alpha=f(\alpha_1,..\alpha_m)$ and $=g(\beta_1,..,\beta_n)$. Let $M_1=\{\alpha_1,..,\alpha_m\}$ and $M_2\{\beta_1,..,\beta_n\}$. Then $\alpha >_T \beta$ iff

T21 f > g and $\alpha >_T \beta_i$ for all i, $1 \le i \le n$, or

T22 f ~ g and

if f and g have left-right status then

there exists j such that $\alpha_i \sim \beta_1,...\alpha_{j-1} \sim \beta_{j-1}$, $\alpha_j > \beta_j$ and $\alpha > \beta_i$ for $j+1 \le i \le n$

if f and g have right-left status similarly

if f and g have no status then

$MP(M_1-M_2) >>_p MP(M_2-M_1)$,

or

T23 f, g are incomparable, and $MP(\alpha) >>_p MP(\beta)$

Note, path comparisons done during term comparisons may require more term comparisons.

This method has not been implemented. It could be implemented and only invoked where RPO failed.

Examples

Examples 1) - 3) follow from RPO.

4) With operator ranking, b > c and no status.

α : f(a(x),g(x,y),b(y)) β : f(x,g(x,y),c(y))

Since the top level function symbols are the same, we must compare the multisets of all full paths in the immediate subterms, or $MP(M_1-M_2) >> MP(M_2-M_1)$

MP(M1-M2) = ({ <a, a(x)>x, <b,b(y)>y, })

and

$MP(M_2-M_1)$ = ({ x, <c, c(y)>y })

Clearly <a, a(x)>x $>_P$ x.

Since b > c, <b, b(y)>y $>_P$ <c, c(y)>y.

Limitations

Once again, there are terms the ordering cannot compare, for example:

α=f(a(x),b(x)), β=g(h(x)) with a>g and b>h

Kapur et al show that it can be determined whether $\alpha >_T \beta$, or $\alpha \sim \beta$ in time $O(|\alpha|^5 * |\beta|^5)$. For example 4 above, this will be (worst case) $O(8^5 * 7^5)$! The method may not be very efficient, but then neither may the others - no complexity analysis of them could be found.

Other Methods

The recursive decomposition ordering [Lescanne 83] (RDO) differs from the recursive path ordering in that it first processes the terms to build their decompositions, determining which symbols, called leaders, are significant according to their position in the term and the given precedence. The decompositions are then compared. It is similar to path ordering in that all paths in a term and all the tuples in a path are taken into account, yet there are terms which can be ordered by path ordering, but not by RDO.

The main advantage of the decomposition ordering is that when it fails to order terms it can suggest an enlargement of the precedence - a set of ordered pairs of symbols extracted from the leaders of the terms. Thus the precedence can be built up step by step. The monotonicity of the precedence ordering is maintained - a new pair may be added, but none are removed or changed.

In the REVE system, RPO and RDO are used together, if the recursive path ordering fails, then the recursive decomposition ordering is called to suggest an enlargement of the precedence.

Other methods have also been suggested, such as polynomial orderings. No one ordering can ever be satisfactory, since the problem is undecidable.

4.1.4 CONFLUENCE

Confluent term rewriting systems mean that replacements may be effected deterministically with no need to backtrack to consider other rewritings. This is equivalent to the Church-Rosser property which expresses the fact that interconvertibility of two terms can be checked by mere simplification to a common form [Huet 80 p797].

A term rewriting system is locally confluent if for any A which reduces to B or C by using a reduction once in each case, there exists a D such that B and C reduce to D (also called the lattice condition). Confluence can be deduced from local confluence if the system is uniformly terminating.

Example - Group Theory

Given a set of equations, the corresponding set of rules is not always convergent. For the equations for the identity operator e, the inverse operator -, * and /:

```
x * e = x
x * x- = e
(x * y) * z = x * (y * z)
x / y = x * y-
```

the corresponding set of rules

```
x * e -> x
x * x- -> e
(x * y) * z -> x * (y * z)
x / y -> x * y-
```

is not convergent: e and x * (y * (x * y)-) are two irreducible terms obtained by rewriting the term (x * y) * (x * y)-.

The Knuth-Bendix Completion Algorithm, Critical Pairs and Superposition

The Knuth-Bendix completion algorithm attempts to transform a set of equations into an equivalent convergent term rewriting system. The term rewriting system is equivalent if it proves the same theorems as a method based on the original set of equations, i.e it forms a decision procedure for the original equational theory.

Confluence is implied by the confluence of certain special cases: critical pairs. Critical pairs are computed by the superposition algorithm [Huet 80 p809], where an attempt is made to match in the most general way the left-hand side of a rule with a non-trivial subterm of another left-hand side.

Let $\alpha 1 \rightarrow \beta 1$ and $\alpha 2 \rightarrow \beta 2$ be two reductions in R. Let M be a non-trivial subterm in $\alpha 1$, such that M is unifiable with $\alpha 2$. Let N be the term obtained when unifying M and $\alpha 2$ such that $Var(N) \cap Var(\alpha 1) = \{\}$. $\delta 1$ is the set of substitutions required to match N with M. $\delta 2$ is the set of substitutions required to match N with $\alpha 2$.

The superposition of $\alpha 1 \rightarrow \beta 1$ on $\alpha 2 \rightarrow \beta 2$ determines the critical pair (P,Q) defined by:

$$P = \alpha 1 \delta 1 \; [M \leftarrow \beta 2 \delta 2]$$
$$Q = \beta 1 \delta 1$$

Two trivial cases may be omitted in finding the critical pairs -

1 when $\alpha 1 = \alpha 2$, and $M = \alpha 1$,

2 when M is a nullary operator.

Here, as elsewhere, the set of variables in each term being unified must be disjoint.

Extension to a complete set

When a set of reductions is incomplete, it may be possible to add further reductions to complete the set, checking after each addition that the new reduction preserves termination. After adding these new pairs, some pairs may become reducible and can be removed.

The algorithm to accomplish this accepts as input a set E of equations, and an ordering > on terms. It produces a convergent term rewriting system, R, or terminates with an error message.

Completion Algorithm

R = {}.

Repeat as long as there are equations in E, if none remain terminate successfully.

1. Remove an equation s=t from E. If R(s)=R(t) repeat step 1.
2. Order the pair (s,t) using >. If the ordering fails, then terminate with an error message, else suppose s > t.
3. Add the rule s->t to R.
4. Use s->t, and the other rules in R to reduce the right-hand sides of the existing rules to normal form.
5. Add every critical pair (P,Q) of terms formed from R using
   s->t to E.
6. Remove from R all old rules whose left-hand side contains an instance of s.

[Hermann 86 p148]

There is no guarantee that this process will terminate. It may be possible to detect the cases where an infinite set of rules will be generated using the concept of a crossed pair of rules. While it can be shown that non-termination is connected with the ordering of symbols where crossed rules are present in some examples, there exist counter-examples where an infinite set of rules will be generated regardless of the ordering [Hermann 86].

Example - The Inverse Property

Suppose we have two operators

. : sym, sym -> sym

- : sym -> sym

and the single axiom

a-.(a.b) = b

First superpose the left-hand side a-.(a.b) onto the non-trivial subterm a.b obtaining (after renaming variables):

M = a1.b1

N = a1-.(a1.b1)

$\delta 1$ = { (a1-,a1), ((a1.b1),b1) }

$\delta 2$ = {}

P = a1--.b1

Q = a1.b1

Both P and Q are irreducible. Since P > Q, add the rule

a--.b -> a.b

to the set.

Now superpose a-.(a.b) -> b on a--.b -> a.b obtaining:

M = a1-.(a1.b1)

N = a2--.(a2-.b2)

$\delta 1$ = { (a2-,a1), (b2,b1) }

$\delta 2$ = { (a2-.b2,b2) }

P = a2.(a2-.b2)

Q = b2

Again another rule is added,

a.(a-b) = b

and by checking these 3 rules the set is shown to be complete.

Example - Group Theory

Using the axioms given for group theory above, and the completion algorithm, this convergent rewrite system is generated with precedence e < * < - < / , and left-to-right status for * :

```
x * e -> x
e * x -> x
x * x- -> e
x- * x -> e
(x * y) * z -> x * (y * z)
x * (x- * y) -> y
x- * (x * y) -> y
e- -> e
x-- -> x
(x * y)- -> x- * y-
x / y -> x * y-
```

If, however, / rather than * plays the main role, this system results with precedence e < - ~ / < * , and right-to-left status for / :

```
x / e -> x
e / x -> x-
x / x -> e
x / (y / z) -> ( x / (y-)) / z
(x / y) / y- -> x
(x / y-) / y -> x
e- -> e
x-- -> x
(x / y)- -> y / x
x * y -> x / (y-)
```

4.1.5 PROVING THEOREMS

To prove a theorem, reduce the left-hand and right-hand sides by applying the rules non-deterministically. If the resultant terms are the same, then the theorem holds.

Example - Inverse Property

The theorem

$$x * (x^{---} * y) = x * y$$

holds since the left-hand side reduces to $x * (x^{-} * y)$ by rule 2, and then $x * y$ by rule 1.

The theorem

$$x * (x^{---} * y) = y$$

does not hold since $x * y$ is not equal to y.

Proof of Inductive Theorems

Huet & Hallot developed a method of proving inductive theorems as follows: Add the statement to be proved to the given convergent set and try to generate a new convergent set while checking that a few simple form conditions are satisfied. If the algorithm succeeds, the statement is a theorem. If it fails by generating a forbidden equation (eg a relation between the constructors), the statement is not a theorem. If it does not terminate, nothing can be proved either way [Lescanne 83].

4.1.6 SHORT-CUTS AND SYNTHESIS

Linearity

A term α is linear if all variable occurrences in α are distinct. A term rewriting system is left linear (respectively right-linear) iff for all $\alpha \rightarrow \beta$ in R, α (respectively β) is linear. Linearity is shown to guarantee that combining two term rewriting systems yields a terminating system [Dershowitz 81]. This could prove handy in optimising the Larch implementation since combining specifications occurs frequently.

Synthesizing implementations from Rewrite Rules

[Kapur, Srivas 85] give a formal completely automatic method for synthesizing Lisp implementations based on convergent term rewriting systems - reversing the process of reduction (ie expansion). Both the type being implemented and the representing type (eg arrays) are given as abstract types.

See Appendix C and [Hsiang, Srivas 85] for research into a method for synthesizing Prolog implementations from the rules. The method is unable to handle the constructor operations, it treats them as symbols. The user must decide on the representation and code it herself. [Bouge 85] describes how the resultant Prolog program can be used to generate test sets.

4.2 DESIGN AND SPECIFICATION

4.2.1 THEOREM PROVING

Design Principles

The theorem prover is the "heart of any implementation of the Larch Shared Language" [Guttag, Horning, Wing, p42]. Yet most of the properties to be checked may be undecidable - "the best any checker can do is to answer 'definitely OK', 'definitely bad', or 'too hard'" [Guttag, Horning, Wing 85 p42].

The guiding design principles were therefore to build a powerful theorem prover which was able to

* solve as large a class of solvable problems as possible,
* provide helpful feedback to the user if the theorem proved false,
* provide an explanation if it was unable to prove a theorem,

  or unable to build a set of reduction rules.

[Lescanne 83 p102] emphasizes how critical uniform termination is: "uniform termination is an important and often neglected aspect of proof methods based on rewriting". The theorem prover is designed with this in mind, and uses more than one method of ordering terms.

The theorem prover operates without user intervention, since it cannot be expected that the user have any knowledge of term rewriting systems. However the theorem prover does make

assumptions on the part of the user according to how they have presented their specification, eg the order operators are presented in **introduces** and the **generated by** list. These assumptions are reflected in the rank assigned operators and the ordering of the axioms in the output file.

Efficiency in terms of speed and space were secondary to the aims of solving as large a class of theorems as possible and of providing useful output to the user.

The theorem prover utilises the data structures built by the parser: operator trees, expressions, axiom list and the like.

4.2.2 SUBSTITUTION, UNIFICATION AND REDUCTION

Bindings and Environments

The **environment** is a list of **bindings**, represented as pairs of variables and expressiontrees, with each new binding appended to the front of the list.

```
Binding : trait
includes Variable, ExpressionTree,
         Pair with [{$ #;# $} for < #;# >,
                    {! #;# !} for T1, ExpressionTree for T2,
                    .var for .first, .expr for .second ]

Environment : trait
includes Binding, LList with [ {$ #;# $} for item,
                               Environment for LList ]
```

Substitution

Substitution applies the bindings to an expressiontree.

```
Substitution : trait
includes Binding, Environment, ExpressionTree
introduces
     subs : Environment, ExpressionTree -> ExpressionTree
asserts for all [ v : Environment, b : Bindings, e :
expression ]
     subs(newlist,e) = e
     subs(insert(v,b),e) =
          if isnewexpr(e) then e
```

```
else
   if isleaf(e) then
       (if (content(e) = b.var) then
           subs(v,b.expr)
        else subs(v,e))
    else subs(insert(v,b),
          enterexpr(subs(insert(v,b),e.leftexpr),
          content(e),subs(insert(v,b),e.rightexpr))
```

Disagreements and unification

Disagreement identifies the point at which two expressiontrees disagree.

Unification builds up the list of bindings required to unify two expressions which are Unifiable. Lookfor, included in unifiable, is true if a variable in one of the disagreeing expressions does not occur in the other. It is not specified here.

There are two issues involved in unification: can the two expressions be unified, and, if so, how. This requires two operations, one (**canunify**) to answer the first question, and the other (**unify**) to answer the second. This problem re-occurs in the next few sections, and is treated in the same manner.

```
Disagreement : trait
includes ExpressionTree,
     Pair with [ExpressionTree for T1, ExpressionTree for T2 ]

Disagree : trait
includes ExpressionTree, Disagreement
introduces
     CanDisagree : ExpressionTree, ExpressionTree -> Bool
     Disagree    : ExpressionTree, ExpressionTree ->
Disagreement
asserts for all [ e1, e2 : ExpressionTree ]
     CanDisagree(e1,newtree)
     CanDisagree(newtree,e2)
     CanDisagree(e1,e2) =
```

```
      if content(e1) = content(e2) then
         candisagree(e1.leftexpr,e2.leftexpr)
            ¦ candisagree(e1.rightexpr,e2.rightexpr)
      else false
     Disagree(e1,newtree) = <e1;newtree>
     Disagree(newtree,e2) = <newtree;e2>
     Disagree(e1,e2) =
      if content(e1) = content(e2) then
        (if disagree(e1.leftexpr,e2.leftexpr) =
          <newtree;newtree>
            then disagree(e1.rightexpr,e2.rightexpr)
            else disagree(e1.leftexpr,e2.leftexpr))
      else <e1;e2>

Unifiable : trait
includes Binding, Disagreement, VariableTree
introduces
     IsUnifiable : Disagreement -> Boolean
     Unifiable   : Disagreement -> Binding
asserts for all [ d : Disagreement, b : binding,
                  vtree: VariableTree ]
     unifiable(d) =
       if findvar(vtree,content(d.first))
         & ~lookfor(content(d.first),d.second)
            then {$ content(d.first);d.second $}
       else
       if findvar(vtree,content(d.second))
         & ~lookfor(content(d.second),d.first)
            then {$ content(d.second);d.first $}
       else {$ newtext;newexpr $}
```

```
Unification : trait
includes ExpressionTree, Environment, Unifiable, Substitution
introduces
     canunify : ExpressionTree, ExpressionTree, Environment
                    -> Boolean
     unify    : ExpressionTree, ExpressionTree, Environment
                    -> Environment
asserts
for all [ e1, e2 : ExpressionTree, v : Environment ]
     unify(e1,e2,v) =
          if ~ canunify(e1,e2) then newlist
          else
          if disagree(e1,e2).first = newexpr then v
          else
          unify(

          subs(insert(newlist,unifiable(disagree(e1,e2))),e1),

          subs(insert(newlist,unifiable(disagree(e1,e2))),e2),
            insert(
              subs(
                insert(newlist,unifiable(disagree(e1,e2))),v),
               unifiable(disagree(e1,e2)))
               )
```

If the expressions cannot be unified, exit with the value newlist. If there is no disagreement, unify has succeeded with the environment v.

In other cases, the **binding** generated by **disagree** must be applied to the two expressions, and applied to the environment generated so far, and then added to the environment so far.

These then form the parameters for the next evaluation of **unify**.

Reduction

Reduction reduces an expressiontree with respect to the given list of rules, until an irreducible expression is obtained. **Rules** and rulelists are specified using axioms and axiomlists, since they have the same structure. Obviously empty expressiontrees or variables cannot be reduced.

A special case of Unify, **Match**, is required. It uses Matchable instead of Unifiable. Matchable is less general then Unifiable: the first expression of the disagreement (the sub-expression of the rule) must be trivial, ie a variable or function with arity of zero. This is to prevent altering the rules and the possibility of infinite reductions.

```
Reduction : trait
includes
     ExpressionTree, RuleList, VariableTree,
     Substitution, Match
introduces
     canreduce : RuleList, ExpressionTree    -> bool
     reduce    : RuleList, ExpressionTree    -> ExpressionTree
asserts for all [ l : RuleList, r : Rule, e: ExpressionTree,
                  vtree: VariableTree ]
     reduce(newlist,e) = e
     reduce(insert(l,r),e) =
       if isnewexpr(e) ¦ findvar(vtree,content(e))
       then e else
           if canmatch(r.lhs,enterexpr(
```

```
        reduce(insert(l,r),e.leftexpr),e,
        reduce(insert(l,r),e.rightexpr)))
    then reduce(insert(l,r),
        subs(match(r.lhs,enterexpr(
          reduce(insert(l,r),e.leftexpr),e,
          reduce(insert(l,r),e.rightexpr),
            r.rhs))
    else reduce(l,e)
```

The specification does not take into account the labelling of variables to avoid conflicts which is required in the implementation.

4.2.3 UNIFORM TERMINATION

Uniform Termination

There are various means for ordering terms. This specification describes Recursive Path Ordering, in its simplified form without operator statuses. We only consider binary expressions.

```
ExprGreater : trait
includes VariableTree, Cardinal
introduces
     ExprGreater    : ExpressionTree, ExpressionTree -> bool
asserts for all [e1, e2 : ExpressionTree]
     ExprGreater(e1,e2) =
          if isnewexpr(e1) | isnewexpr(e2) then false else
%         CASE 1
          if ~isleaf(e1) & isleaf(e2) then
            findvar(Vtree,content(e2).opid)
              and ContainsVar(e1,content(e2).opid)
          else
          if ~isleaf(e1) & ~isleaf(e2) then

%         CASE 21
          (if content(e1).opweight > content(e2).opweight then
            ExprGreater(e1,e2.leftexpr) &
            ExprGreater(e1,e2.rightexpr))
          else
%         CASE 22
          if content(e1).opweight = content(e2).opweight then
            MultiGreater(e1.leftexpr,e1.rightexpr,
```

```
                    e2.leftexpr,e2.rightexpr)
          else
%         CASE 3
               ExprGreater(e1.leftexpr,e2)
               | ExprGreater(e1.leftexpr,e2)
          else
               false
```

ExprGreater has value true if the first expressiontree is "greater" than the second. It makes use of the operator > from **Cardinal**, and the operator ContainsVar which determines if a given variable occurs in an expression. Multigreater is the multiset extension.

4.2.4 CONFLUENCE AND CRITICAL PAIRS

Completion

Again, there are two issues: can a set of axioms be completed, and, if so, how? **Cancomplete** addresses the first problem, **Complete** the second.

Normalise takes a rulelist and reduces the right-hand sides to normal form while **Shrink** removes rules from the list whose left-hand sides contain an instance of an Expressiontree.

Cpairs (critical pairs) adds the list of new axioms (P,Q) formed by the operation superpose to the current axiomlist. **Superposition** requires not only the environment for matching two terms, but the resulting expression. The specification assumes the operator UnifiedExpr provides this.

```
Completion : trait
includes ExprGreater, AxiomList, RuleList, Unification
introduces
     normalise      : Rulelist                        -> RuleList
     instance       : Expressiontree, Expressiontree   -> bool
     shrink         : Rulelist, ExpressionTree        -> Rulelist
     superpose      : Rule, ExpressionTree            -> Axiom
     cpairs         : Rulelist, Axiom                 -> AxiomList

     cancomplete    : AxiomList, Rulelist             -> bool
     complete       : AxiomList                       ->  RuleList
```

```
asserts for all [al : axiomlist, a : axiom,
                 rl : rulelist,  r : rule, e : ExpressionTree]

     shrink(newlist,e) = newlist
     shrink(insert(rl,r),e) =
          if newexpr(e) then insert(rl,r)
          else if instance(e,r.lhs) then shrink(rl,e)
          else insert(shrink(rl,e),r)

     normalise(newlist) = newlist
     normalise(insert(rl,r)) =
          if canreduce(rl,r.rhs) then
               insert(normalise(rl),reduce(rl,r.rhs))
          else
               insert(normalise(rl),r)

     superpose(r,e) =
          if nontrivial(e) and canunify(r.lhs,e) then
               {! subs(unifiedExpr(r.lhs,e),e);
                  subs(unify(unifiedExpr(r.lhs,e)),r.rhs) !}
          else
               newaxiom

     cpairs(newlist,a) = newlist
     cpairs(insert(l,r),a) =
          insert(insert(insert(
               cpairs(l,a),superpose(r,a.lhs.leftexpr)),
                           superpose(r,a.lhs.rightexpr)),
                           superpose(r,a.lhs))

     cancomplete(newlist,rl) = true
     cancomplete(insert(al,a),rl) =
```

```
if a.lhs=a.rhs then
     cancomplete(al,rl)
else
if ExprGreater(a.lhs,a.rhs) then
     cancomplete(
  appendlist(al,cpairs(
    normalise(insert(rl,{! a.lhs;a.rhs !})),a)),
      shrink(
        normalise(
         insert(rl,{! a.lhs;a.rhs !})),a.lhs))
else
if ExprGreater(a.rhs,a.lhs) then
     cancomplete(
  appendlist(al,cpairs(
    normalise(insert(rl,{! a.rhs;a.lhs !})),a)),
      shrink(
        normalise(
          insert(rl,{! a.rhs;a.rhs !})),a.rhs))
else
     false
```

Complete follows from cancomplete.

4.2.5 SEMANTIC CHECKING

Consistency

A traitbody is consistent if its associated theory does not contain the equation false=true.

This is checked while completing the set of reduction rules. If the critical pair (true,false) is generated, where true and false are the constructors from the standard trait boolean, then the axiom system is inconsistent. If the pair (true,false) is generated while trying to prove a theorem, then either the system is inconsistent, or the theorem is false.

Assumptions

Let A(T) be all the assumes of the traits imported or included in T, and R(T) be the result of translating T after removing these assumes. A(T) is discharged by T if the theory associated with the translation of each **traitRef** of A(T) is a subset of the theory associated with R(T).

This check is accomplished by marking the axioms which are indirectly assumed in a trait which has been imported or included. They can then be removed from the set of axioms from which the rewriting system is generated. The marked axioms should now be provable. As a shortcut, axioms from traits which are assumed, but later explicitly included or directly assumed, can be removed off the list (this is frequently the case).

Example

A trait could be specified for **OrderedSets** which either makes no assumptions about the < operator, or assumes the trait **TotalOrder** to define it as a total ordering for example. If the trait for **OrderedSets** is specialized:

```
IntOrderedSets : trait
     includes OrderedSets with [ Integer for Elem]
     imports Integer
```

then **TotalOrder** must be shown to be a subset of the theory associated with **IntOrderSets.**

In practice, **Integer** will either explicitly include **TotalOrder**, or the trait **IntOrderedSets** will assume **TotalOrder** to discharge the assumption of the trait it has included.

Imports

The theory associated with a trait must be a conservative extension of the theory associated with the translation of each trait in its **imports**; i.e. if trait T1 imports trait T2 and W is a well formed formula (wff) containing only operators introduced in T2, W is in the theory associated with T1 iff it is in the theory associated with T2.

To check this, mark all the relevant axioms in T which use only operators from an imported trait. These axioms can then be removed from the set used to generated the rewrite rules. These marked axioms should then be provable.

Operators appearing in an imported trait may not be constrained by the importing trait or any other imported trait. This guarantees that imported traits don't interfere with one another in unexpected ways. This check is easily accomplished during parsing and does not require any theorem proving.

Example

All traits import the standard trait boolean, and can't suddenly change the meaning of true and false, or any of the logical operators. If a trait has a wff b & b = false, then it would not be semantically correct, since b & b = false is not in the theory for boolean. (b & b = true is in the theory, but true = false is not).

Constraints

A **proppart** is properly constraining if it implies properties of only the operators in its **constrains**. If T is a trait, and P is the proppart

constrains { SortId | SortedOp*, } so that props

then the trait T plus P is properly constraining if each wff in P is also in the theory associated with T, or else contains a **sortedOp** listed. The occurrence of a **sortId** in a constrains indicates that all operators whose signature includes that **sortId** are constrained.

The check is done only on traits in which constrains appears

explicitly. When the trait is included in another, the **constrains** is changed to **asserts**.

Once again, simply mark the relevant axioms, remove them, generate the rewrite rules, and then check that they hold.

A trait may have the form: T P1 P2 P3 where P1, P2 and P3 are separate **propparts** with **constrains**. Only the axioms from T should be used when checking P1, T+P1 when checking P2, and T+P1+P2 when checking P3. This is achieved by building the reduction rules and performing the semantic checks at the end of each **constrains**.

Example

Ordinal is a handbook trait specifying the operators first and succ on sort Ord. The trait **Cardinal** constrains the operator 1:

```
Cardinal : trait
imports Ordinal with [ 0 for first, Card for Ord ]
introduces
     1: -> Card
constrains 1 so that
     1 = succ(0)
```

This means that the theorem 1 = succ(0) should be provable using the axioms from **Ordinal**, or else contain the operator 1.

Consequences

A trait implies its **consequences** if the theory associated with its **conseqProps** is a subset of the theory associated with the trait; and the **sortedOp** in each **converts** is convertible. Implies is used to indicate intended consequences of the specification, both for checking and to increase the reader's insight. Convertibility is defined using the theory and the exempts of a trait, and is used to indicate that the **SortedOp** is adequately defined (the "completeness" property).

The theory associated with the **conseqprops** is easily checked to be a subset by attempting to prove the theorems as presented in the consequences using the rewrite system generated from the axioms.

Examples

If the standard trait **boolean** included these consequences

```
implies asserts for all []
     (false & (true & false)) = ~ (~ false))
```

then reduce the left-hand side obtaining false, and reduce the right-hand side obtaining false. Since false=false, the theorem holds.

The handbook trait for the generic **Join** of two containers

```
     implies Associative with [ .join for o, C for T]
```

This means the theorem from Associative

```
     (x .join y) .join z = x. join (y .join z)
```

must hold.

Checking for **converts** is formally defined as follows in the reference manual [Guttag, Horning, Wing 85 p59]: Let C be a **conversion**. For each term, t, that contains no variables of any sort appearing in a generators in the containing trait, the theory of the containing trait must either

* contain an equation $t=t_1$, where t_1 contains no **SortedOp** appearing in C's **SortedOp***, or
* contain an equation $t'=t_1$, where t' is a subterm of t, and t_1 is an instantiation of a term appearing in an exempts of the containing trait.

This definition is unclear and confusing. The informal explanation of Larch [Guttag, Horning, Wing p34-35] makes it clear that t is not any term, but a term of the form

SortedOp { '(term*, ') }

where the SortedOp is in the conversion C. Converts is intended to check that the axioms adequately define an operator. It's use can best be demonstrated by an example.

Example

```
stack : trait
introduces
     new  :          -> stk
     push : stk, el  -> stk
     pop  : stk      -> stk
     top  : stk      -> el
     isnew: stk      -> bool
asserts
stk generated by [new, push]
```

```
stk partitioned by [isnew, pop, top]
for all [s : stk, e : el]
     isnew(new)
     ~isnew(push(s,e))
     pop(push(s,e)) = s
     top(push(s,e)) = e
converts [isnew, top, pop]
exempts
     pop(new),
     top(new)
```

The terms to be checked are:

```
isnew(new), isnew(push(new,e)), isnew(push(push(new,e),e)...
top(new), top(push(new,e)), ...
pop(new), pop(push(new,e)), ...
```

(Note: isnew(s), top(s), pop(s), ... are excluded since stk appears in the generators)

isnew(new) and isnew(push(s,e)) clearly satisfy the first criterion. top(new) and pop(new) satisfy the second for exempt terms. top(push(new,e)) and pop(push(new,e)) can both be reduce to new and thus satisfy the first criterion. It is not necessary to check further terms such as isnew(push(push(new,e),e)) since all stacks are generated by the terms push and new, and induction on these terms is part of the theory associated with the trait. new is used as the basis, and push(s,e) in the induction step.

This semantic check is the most complex. It is implemented by generating the term to be checked, checking that it meets one

of the two criteria, then continuing to the next term. The process terminates when there are no more terms to be checked (as in the above example), or when a term fails to meet one of the criteria. It is possible that the process could continue ad nauseam, for example if the **generators** had been ommited in the stack specification above. Precautions against this need to be incorporated.

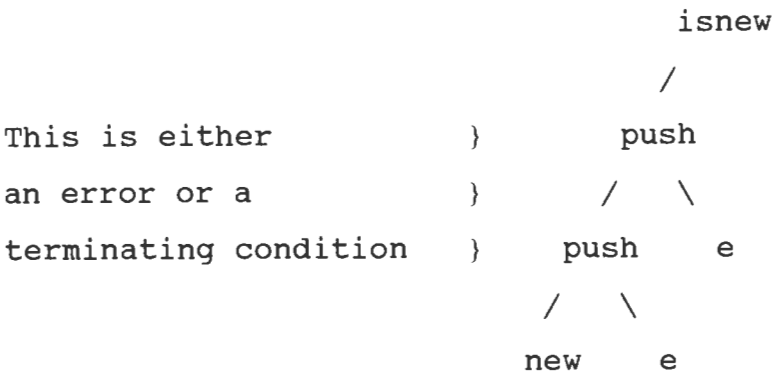
Before generating the terms a matrix of sorts is built, where each column is a particular sort, and each element in that column is either a variable, or an operation whose range is that sort. The columns are ordered with the variable first, then the operations from lowest to highest arity. The variable is removed if the sort appears in a generators.

The matrix for the stack trait would be

Sorts:	stk	el
	new	e
	push	

This matrix is used to generate the terms to be converted: the first element in each column in used to generate the first term, eg isnew(new,e). Subsequent terms are recursively generated from the previous term with reference to the matrix, eg isnew(push(new,e),e).

The paths in the tree-form of the term are checked for repetition, eg isnew(push(push(new,e),e)), to avoid infinite generation of terms.


isnew
This is either
an error or a
terminating condition
push
push
e
new
e

Specification

Semantic checks require the reduction and unification traits, theorems and rulelists. They would only be used once a list of confluent reduction rules had been built from the axioms.

Consistency is dealt with in the process of finding a confluent set of reduction rules.

A list of the assumed rules is built by the trait BuildAssumes (similar to parse, but with RuleList as its range). For the assumptions to be discharged, these rules must be provable.

```
Assumptions : trait
imports RuleList, Reduction
introduces
     assume    : RuleList, RuleList     -> Bool
asserts for all [ a, l : RuleList, r : rule ]
     assume(l,newlist) = true
     assume(l,insert(a,r)) =
          (reduce(l,r.rhs) = reduce(l,r.lhs)) & assume(l,a)
```

Imports, constraints and consequences can be checked in a similar way.

SemanticCheck is a trait which combines all the above.

```
SemanticCheck : trait
includes Assumptions, Imports, Constraints, Consequences
introduces
     SemanticCheck  : RuleList     ->   Bool
```

```
asserts for all [l, a, i, c : Rulelist]
     SemanticCheck(l,a,i,c) =
          Assume(l,a) & Import(l,i) & Constrain(l) &
Conseq(l,c)
```

4.2.6 IMPLEMENTATION DETAILS

Labelling and Unlabelling

Many of the routines in the theorem prover (eg critical pairs, unification) involve two expressions, eg a-.(a.b) and a.b. It is necessary to distinguish between the variable a which occurs in both expressions. This is done by labelling the variables in each expression, a simple method would be a#1-.(a#1.b#1) and a#2.b#2. (# is a useful character, since the user cannot possible use it in an identifier).

Once the routine is complete, the labels must be removed. This produces problems if the resulting expression contains a#1 and a#2, a quite frequent occurrence in the critical pair routine. Conflicts like this can be resolved by keeping a record of all the labelled variables and the corresponding unlabelled variable, eg a#1 and a, and generating a new unique variable where a conflict arises, eg a#2 and a1.

Garbage collection

Each call to unify two expressions, or to attempt to reduce an expression requires a copy of the expression to manipulate. Space is allocated on the heap, but efficient garbage collection is essential [Kahn 85].

Optimization

Optimizations to the completion algorithm are:

* testing "short" reductions first - these are more likely to lead to interesting consequences which cause "long" reductions to reduce or disappear. ("Short" reductions are reductions short in length, or short in weight). [Knuth Bendix 83]. This has been done by keeping the axiom and reduction rule lists sorted.
* coding the most heavily used subroutines in assembly language. This has not been done.

CHAPTER 5 : CONCLUSIONS AND AREAS FOR FURTHER RESEARCH

The aim of this project was to prototype formal specification in Larch, in particular to:

* Implement the Larch Shared Language
* Write a non-trivial specification in Larch.
* Employ formal methods in the implementation process.

Towards this aim I have :

* Built a parser and context sensitive checker for the Shared Language.
* Implemented term rewriting methods for proving theorems.
* Developed methods for performing semantic checks using the theorem prover.
* Provided a specification development aid in the form of structured output describing the relationships and elements in a specification and listing the resulting reduction rules.
* Written a specification for this implementation of Larch in Larch.
* Used the specification as a basis for formal development.

5.1 THE LARCH SHARED LANGUAGE

The Implementation

As it stands, this implementation of Larch is useful as a tool for the writing and testing of Larch specifications, but could benefit from an editor. A library manager would be useful in cases where a team was developing a system, for maintaining links between dependent sepecifications.

Evaluation

The Larch implementation is ideally suited to formal specification - the requirements are clearly stated in advance and static, the data structures are straightforward, the user interface is minimal and doesn't require specfication, there are no peripheral devices are any other hard-to-specify areas. The specification was only used for initial development, no subsequent maintenance was done. The comments that follow should be read with this in mind.

Larch is restricted to the algebraic style of specification. This simplicity makes it fairly easy to learn and use, but is limiting if the project has components which cannot be expressed algebraically, yet can be expressed in other formalisms (such as the chosen programming language!).

While writing the specification, I seldom had difficulty expressing a data structure (though only simple ones were used), but sometimes had difficulty expressing an algorithm (very complicated one were used!). This was not necessarily a

result of any intractability of Larch, it could have been due to difficulty in finding the appropriate abstractions and the inherent complexity of the algorithm.

The greatest initial difficulty encountered was avoiding thinking in terms of coventional sequential programming constructs. With practise, this difficulty is overcome.

A specification technique should encourage the writer to think in terms of external behaviour, not in terms of internal component details. The Larch Shared Language achieves this.

A useful specification should help organize the information to be presented. The Larch specifications in Chapters 3 and 4 rely on natural language, white space, and ommissions of repetitive detail to make it presentable. Larch organizes the information on a small scale at trait level, but lacks a means to give the user an overview, a graphical picture, cross references and other devices used to organize information on a large scale.

The resulting specifications are very readable to the trained person who wrote them, but probably obscure to an untrained user. The level of training required to understand Larch, particularly the semantic checks, is high. It is possibly too high to justify its worth in the fast-moving world of industry, particularly in South Africa where even simple structured programming skills are in short supply. Larch is merely a language, it is not a method, and provides no guidance, other than examples and experience, for writing specifications.

A Larch shared langauge specification can be useful as intermediate documentation for an implementor since it has the essential properties of clarity and precision, without constraining the implementation. It has also proved useful as a tool for discussion and argument. However, the specification is useless as a contract between the system designer and the system user, since it is not equally understandable by both. An alternative representation mechanically derived from the specification may be more helpful to an untrained user - such as a User Manual. It does not provide support for documenting design decisions (other than comments), and there are no facilities for version control.

The handbook was useful. Traits in it and others in Chapters 3 and 4 (Trees, Lists), were easily reused and modified as needed using the **with**-list. However, reusable traits do not arise without some effort in their design. The "putting together" operations are a strength of the language, and essential for constructing any specification longer than a page or two.

Some of the specifications given were submitted to the parser and semantic checker. The parser was useful in finding syntactic errors, such as missing parameters. The more complicated semantic checks were useful in proportion to the effort put in by the specifier - incompleteness or inconsistencies can only be detected if sensible theorems are listed in the **implies**, or terms are listed in the **converts**.

5.2 SPECIFICATION LANGUAGES

More research is needed into mechanized support tools, not only in the design and building of such tools, but also in investigating their impact. Examples of such tools have been listed in Chapter 2. High on the list is likely to be mechanisms for multiple views from the various viewpoints of designer, programmer, client or other interested parties. The output file mentioned above is a simple, but very useful, example.

Larch is useful for specifying the abstract data structures of a system, but other languages may be more suited for specifying other parts of the system. For example, there are many existing methods for expressing the syntax of a language which would provide a more succint, readable and elegant specification than one written using Larch. Research is needed into combining methods and languages - should a "melting-pot" approach be used, where different languages are integrated, or a "salad" approach with the best from each used where appropriate? Which languages combine well and what are the implications for proving consistency? How will the interfaces between specifications in different languages be managed?

More practical experience is needed into the scaling up problem and management of large scale specifications. Often there are insufficient resources for complete specification. The parts of the system which can benefit most from complete formal specification, and the parts of the software lifecycle which can benefit most from rigorous formal methods need to be

identified. For example, the specification for the lexical analyzer and parser might be left out on the basis that the area is well understood and a specification was already provided in the Larch documentation (Appendix A). Yet writing the specification, which was done after coding, had clear benefits - at least one error caused by misreading the grammar was bought to light. It is unlikely the error would have been found while testing the parser, since it is impossible to test a complete set of specifications.

Little work has been done on specification languages for concurrency or for distributed systems, and there are correspondingly few examples of projects where formal specifications have been employed.

Should specifications be executable or not (should recipes be edible)? Does executability help in understanding the specification or hinder, likewise in learning how to specify? One route is to combine formal non-executable specifications with rapid prototyping.

[Arango & Freeman 88] summarize research problems where existing AI techniques can be used. In the longer term, issues which require considerable research include: domain analysis (getting the domain-specific knowledge needed for specifications); performance specifications; model extension by learning (the process of extending models is currently performed in an ad hoc manner but can be formalized); cost-effectiveness (under what circumstances is investment in expensive formalization and knowledge-based approaches justified).

Answers to these questions can only come with more practical experience in the use of specifications.

5.3 FORMAL METHODS BASED ON SPECIFICATIONS

The claims of the formalists can now be examined.

The specification given and the implementation were not developed in well-defined stages, in contrast there was an "ineviatable intertwining of specification and implementation" [Swartout & Balzar 82]. Although the program was developed in a process closer to that described by the non-formalists, the formalism did not hinder the process as they claim, but was a valuable guide. For example, if some part of the program was proving difficult to implement, specifying it helped think about the problem in an abstract way, and often suggested a different approach to implementing a solution.

The program was not derived from the specification, but the greater understanding of the problem as a result of specifying it made implementing much easier. None of the ideas presented in Chapter 4 on synthesising an implementation from reduction rules were employed, but this was mainly because the implementation languague (TURBO C) was inappropriate. Neither was the program built by implementing individual pieces of the specification and then combining them. However, the modules do correspond closely to the specification traits, with functions for each operator. Likewise, the C source files correspond to the subsections of specification (data operations, parser etc). Larch does not aid in determining implementation strategies. Having written the specification, the developer can then ignore it entirely.

The specification helped build a well designed program. One

cannot say if it is a better design since there is no control to compare it to, neither can one easily attribute the credit for a good design to the fact that formal specfication was used (it is more likely a result of fixed static requirements). There is increased confidence in the program's correctness (shown by trial use, not formal proof), but it is likely that any method which required two formal passes at the problem would achieve this [Berry & Wing 85].

The specification was useful for answering questions about the implementation, and was referred to frequently during coding and during testing. No attempt was made to prove the program correct.

In conclusion, the formal methods and formal specification served their purpose as a technique to building a better program.

REFERENCES

Arango.G & Freeman.P,

'Application of Artificial Intelligence' in 'Report on the 4th International Workshop on Software Specification and Design', Software Engineering Notes, Vol 13 no 1, pp 32-40, January 1988

Balzar.R,

'Software Technology in the 1990's : using a new Paradigm' Computer Vol 16 no 11, pp 39-45, November 1983

Balzar.R,

'A 15 year Perspective on Automatic Programming', IEEE Transactions on Software Engineering, vol. SE-11 no 11, pp 1257 -1268, November 1985

Brooks.FR,

'No Silver Bullet: Essence and accidents in Software Engineering', Computer, vol 20 no 4, pp 10-20, April 1987

Balzar.R and Goldman.N,

'Principles of Good Software Specification and their Implications for Specification Languages' in Software Specification Techniques, Gehani.N and MacGettrick.A (eds), 1986

Bergstra.JA and Klop.JW,

'Conditional Rewrite Rules: Confluence and Termination', Journal of Computer and System Sciences, Vol 32 No 3,

pp 323-362, June 1986

Berry.DM and Wing.JM,
'Specifying and Prototyping : some thoughts on why they are successful', in Formal Methods and Software Development - TAPSOFT Proceedings vol 2 1985, Lecture Notes in Computer Science 186 Springer-Verlag

Bouge.L et al,
'Application of Prolog to test sets generation from algebraic specifications', in Formal Methods and Software Development - TAPSOFT Proceedings vol 2 1985, Lecture Notes in Computer Science 186 Springer-Verlag

Burstall.RM and Gougen.JA,
'An Informal introductiuon to Specifications using Clear' in Software Specification Techniques, Gehani.N and MacGettrick.A (eds), pp 363-389, 1986

Chang & Lee,
Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973

Chi.UH,
'Formal Specification of User Interfaces : a Comparison and Evaluation of 4 axiomatic approaches', IEEE trans on Software Engineering SE-11 no 8, pp 671-685, August 1985

Cohen.J,
'Describing Prolog by its Interpretation and Compilation', Comm ACM, vol 28 no 12, pp 1311-1324, December 1985

DeMillo.R.A, Lipton.R.J. and Perlis.A.J,
'Social Processes and Proofs of Theorems', Comm ACM vol 22 no 5, pp 271-280, May 1979

Dershowitz.N,
'Termination of Linear Rewriting Systems', Proceedings Automata, Languages and Programming 8th Colloquium July 1981, Lecture Notes in Computer Science 115, pp 448-457, Springer Verlag

Economist,
'Something rotten in the state of software', The Economist, pp 83-86, January 9 1988

Floyd.C,
'Introduction', in Formal Methods and Software Development - TAPSOFT Proceedings vol 2 1985, Lecture Notes in Computer Science 186 Springer-Verlag

Floyd.C,
'A Paradigm change in software engineering', Software Engineering Notes, Vol 13 no 2, pp 25-38, April 1988

Gehani.N,
'Specifications: Formal and Informal - a Case Study' in Software Specification Techniques, Gehani.N and MacGettrick.A (eds) 1986

Genesereth.MR and Ginsberg.ML,
'Logic Programming', Comm ACM, vol 28 no 9, pp 935-941, September 1985

Guttag.J and Horning.JJ

'Formal Specification as a Design Tool' in Software Specification Techniques, Gehani.N and MacGettrick.A (eds), pp 187-208, 1986

Guttag.JV, Horning.JJ and Wing.JM,
Larch in Five Easy Pieces, Digital Equipment Corporation, 1985

Guttag.JV, Horning.JJ, and Wing.JM,
'The Larch Family of Specification Languages', IEEE Software vol 2 no 5, pp 24-35, September 1985

Hermann.N,
'On NonTermination of the Knuth-Bendix algorithm', Proceedings Automata, Languages and Programming 13th International Colloquium France July 1986, pp 146-156, Lecture Notes in Computer Science 226, Springer Verlag

Henderson.P
'Functional Programming, Formal Specification and Rapid Prototyping', IEEE Trans on Software Enginerring, vol SE-12 no 2, pp 241-250, Febraury 1986

Hoare.CAR,
'An Overview of some formal methods for program design', Computer vol 20 no 9, pp 85-91, September 87

Horning.JJ,
'Combining Algebraic and Predicative specifications in Larch', Lecture Notes in Computer Science 186 - Proc of TAPSOFT 1985, Springer-Verlag

Hsiang.J, Srivas.MK,

'A Prolog environment for developing and reasoning about data types', in Formal Methods and Software Development - TAPSOFT Proceedings vol 2 1985, Lecture Notes in Computer Science 186 Springer-Verlag

Huet.G,

'Confluent Reductions: Abstract properties and Applications to Term Rewriting Systems', Journal of the ACM, Vol 27 No 4, pp 797-821, October 1980

Jacob.RJK,

'Using Formal Specifications in the design of a Human-Computer Interface', Comm ACM Vol 26 no 4, pp 259-264, April 1983

Jones.CB,

'The role of proof obligations in software design', in Formal Methods and Software Development - TAPSOFT Proceedings vol 2 1985, Lecture Notes in Computer Science 186, Springer-Verlag

Kahn.JS,

'Garbage Collection Algorithms', UCT Honours Self-study topic, 1985

Kapur.D, Narendran.P, Sivakumar.G,

'A Path Ordering for proving termination of term rewriting systems', Proceedings TAPSOFT 1985 vol 1, in Lecture Notes in Computer Science 185, pp 173-187, Springer Verlag

Kapur.D, Srivas.M,

'A Rewrite Rule Based Approach for synthesizing Abstract

Data Types', Proceedings TAPSOFT 1985 vol 1, in Lecture Notes in Computer Science 185, pp 188-207, Springer Verlag

Knuth.DE and Bendix.PB,
'Simple Word Problems in Universal Algebras' in Automation of Reasoning 2, Springer-Verlag 1983

Lescanne.P,
'Computer Experiments with the REVE Term Rewriting System Generator', Proc 10th Principles of Prog. Languages, pp 99-108, ACM 1983

Liskov.BH and Berzins.V
'An Appraisal of Formal Specifications', in Software Specification Techniques, Gehani.N and MacGettrick.A (eds), pp 3-23, 1986

Maibaum.TSE,
Veloso.PAS, Sadler.MR, 'A Theory of abstract data types for program development : bridging the gap', in Formal Methods and Software Development - TAPSOFT Proceedings vol 2 1985, Lecture Notes in Computer Science 186, Springer-Verlag

Meyer.B,
'On Formalism in Specifications', IEEE Software Vol 2 no 1, pp 6-26, January 1985

Naur.P,
'Intuition in software development', in Formal Methods and Software Development - TAPSOFT Proceedings vol 2 1985, Lecture Notes in Computer Science 186, Springer-

Verlag

Neumann.P and others,
'Risks to the Public in Computers and Related Systems', Software Engineering Notes, vol 13 no 1, pp 3-16, January 1988

Parnas.DL,
'A technique for software module specification with examples', Comm ACM vol 15 no 5, pp 330-336, May 1972

Pettorossi.A,
'Comparing and Putting together Recursive Path Ordering, Simplification Orderings and Non-Ascending Property for terminating proofs of term rewriting systems', Proceedings Automata, Languages and Programming 8th Colloquium July 1981, Lecture Notes in Computer Science 115, pp 432-447, Springer Verlag

Sunshine.C.A, Thompson.D, Erickson.R, Gerhart.S, Schwabe.D,
'Specification and Verification of communication Protocols in AFFIRM using State transition Models, in Software Specification Techniques, Gehani.N and MacGettrick.A (eds), pp 303-338, 1986

Swartout.W & Balzar.R,
'Ineviatable Intertwining of Specfication and Implementation', Comm ACM vol 25 no 7, pp 438-440, July 1982

Warren.D,
'Logic Programming and Compiler Writing', Software Practice and Experience vol 10, pp 97-125, 1980

Wing.J,

'Writing Larch interface language specifications', ACM Trans. on Prog. Languages and Systems, vol 9 no 1, pp 1-24, January 1987

Woodcock.JCP,

'Formalisms', Software Engineering Notes, Vol 13 No 1, pp 30-32, January 1988

Zave.P,

'An operational approach to requirements specification for embedded systems', IEEE Trans Software Engineering vol 8 no 3, pp 250-269, May 1982

Zave.P,

'The Operational versus the Conventional approach to software development', Comm ACM, vol 27 no 2, pp 104-118, February 1984

Zave.P & Schell.W,

'Salient Features of an Executable Specification Language and its Environment', IEEE Trans, on Software Engineering vol 12 no 2, pp 312-325, February 1986

Zave.P,

'Assessment' in 'Report on 4th International Workshop on Software Specification and Design', Software Engineering Notes, vol 13 no 1, pp 40-43, January 1988

APPENDIX A : LARCH SHARED LANGUAGE REFERENCE

GRAMMAR AND CONTEXT SENSITIVE CHECKING

Character Sets

Specifications in Larch are built up with characters from the alphanumeric character set

A..Z,a..z,0..9

and the punctuation characters

:,#,,,->,=,%,[,],),(,.

User defined variables and sorts may only contain alphanumeric characters. Operators may contain characters from the alphanumeric set, or from the special character set

~,¦,},{,`,_,^,\,!,",$,&,*,+,/,;,<,>,?,@,'

White space characters (space, tab, line-feed, carriage-return, form-feed and new-line) may be used to separate words and lines. Control-Z is treated as an end-of-file marker. Any text following the Control-Z is disregarded.

Syntactic Conventions

|	Alternative Operator
{ e }	e is optional
e*	e may be repeated zero or more times
e*,	e may be repeated zero or more times, separated by commas
e^+	e may be repeated one or more times
alpha	alpha is a non terminal symbol
alpha	alpha is a terminal symbol
'(')	parentheses as terminal symbols
(e)	parentheses for grouping syntactic expressions

Reference Grammar

```
trait          ::=  traitid : trait traitBody {consequences}
                    {exempts}
traitBody      ::=  externals SimpleTrait
externals      ::=  {assumes} {imports} {includes}
assumes        ::=  assumes traitRef*,
imports        ::=  imports traitRef*,
includes       ::=  includes traitRef*,
traitRef       ::=  traitId {renaming}
renaming       ::=  with [ ( sortrename ¦ oprename )*, ]
sortrename     ::=  SortId for OldSort
OldSort        ::=  SortId
oprename       ::=  OpId for OldOp
OldOp          ::=  SortedOp
SortedOp       ::=  opDcl ¦ OpId {-> range}
SimpleTrait    ::=  {opPart} propPart*
opPart         ::=  introduces opDcl*
opDcl          ::=  opId : signature
signature      ::=  domain -> range
domain         ::=  SortId*,
range          ::=  SortId
propPart       ::=  ( asserts ¦ constrains ) props
constrains     ::=  constrains ( SortId ¦ SortedOp*, ) so that
props          ::=  generators* partitions* axioms*
generators     ::=  SortId generated bylist*,
partitions     ::=  SortId partitioned bylist*,
bylist         ::=  by [ sortedOp*, ]
```

```
axioms          ::=  for all [ varDcl*, ] equation*
varDcl          ::=  varId*, : SortId
equation        ::=  term {= term}
term            ::=  secondary ¦ if secondary then secondary else
                     term
secondary       ::=  {opSym} primary ( opsym primary )* {opsym}
primary         ::=  SortedOp { '( term*, ') }   ¦ varId   ¦
                      '( term ')
opId            ::=  alphanumeric+ ¦ opForm
opForm          ::=  {#} opSym ( # opsym )* {#}
opSym           ::=  SpecialChar+ ¦ .alphanumeric+
traitId         ::=  alphanumeric+
SortId          ::=  alphanumeric+
VarId           ::=  alphanumeric+
consequences    ::=  implies conseqprops {converts}
conseqprops     ::=  traitRef*, props
converts        ::=  converts conversion*,
conversion      ::=  [ sortedOp*, ]
exempts         ::=  exempts exemptTerms*
exemptTerms     ::=  { for all [ varDcl*, ]} term*,
```

Comments start with % and terminate with end of line. They may appear after any token.

CONTEXT SENSITIVE CHECKING

Simple Traits:

The sets of VarIds, SortIds, and OpIds must be disjoint. Each SortId and each SortedOp appearing anywhere in the simpletrait must appear in the Oppart.

OpDcl:

If the OpId is an Opform, it must have the same number of #'s as occurrences of SortIds in the signature's domain.

Generators:

The range of each sortedOp must be the SortId of the generators. At least one SortedOp in each bylist must have a domain in which the SortId of the generators does not occur.

Partitions:

The domain of each SortedOp must include the SortId of the partitions. The range of at least one SortedOp in each bylist must be different from the SortId of the partitions.

Axioms:

Each VarId used in a term must appear in exactly one varDcl. No VarId may occur more than once in a [VarDcl*,].

Equation:

The sorts of both terms must be the same, where the sort of the form SortedOp { (term*,) } is the range of the SortedOp, and the sort of a term of the form VarId is the SortId of the VarDcl

in which the varId is declared.

To resolve the grammatical ambiguity between the = connective in equations and the = opSym, the first occurrence of = not bracketed by parentheses or within an if then else is the equation connective; the remainder are opSyms.

Term:

In SortedOp { (term*,) } the domain of the SortedOp must be the sequence of the sorts of the terms in term*,.

ConseqProps:

If the props of the conseqProps is appended to the proppart of the containing trait, the resulting trait must satisfy the checks above.

Exempts:

Each term must satisfy the checks above.

Implicit Signatures and Partial OpForms:

There must be a unique mapping from occurrence of SortedOps to OpDcls of the traitBody such that for each SortedOp, OpDcl pair: the OpIds math, i.e. they are the same, or they are both OpForms and the one in the SortedOp is the same as the one in the OpDcl with all #'s removed; if the SortedOp includes -> range, it is the same as the range of the OpDcl.

Boolean Terms as Equations:

The term must be of the sort bool if the production

equation ::= term is used.

External References:

Recursive externals are not permitted; i.e. the traitId of the containing trait may not appear in an externals, nor in any partial translation of a traitRef in its externals. The translation of a trait is derived bottom-up, i.e. before a trait with traitRefs is translated, each of its traitRefs is replaced by the translation of the trait labelled by that traitRef's traitId.

Let T be a trait with S its simpletrait, and E the translations of the traitRefs in the externals. T consists of:

* an opPart containing both S and E's opDcls
* a propPart containing both S and E's propParts
* a consequences containing the props of T's conseqprops, the propParts of the translations of the traitRefs in T's conseqprops, E's consequences
* an exempts containing both S and E's exempts.

Renaming:

No sortedOp may occur more than once as an oldOp.
No SortId may occur more than once as an oldSort.
Each oldSort must appear in an OpDcl in the translation of the trait labeled by the traitId.
There must be a unique mapping from OldOps to OpDcls of the translation of the trait labeled by the traitId, such that for each oldOp,OpDcl pair: the Opids match; if the OpIds includes a domain, it is the same domain as the domain of the OpDcl, if the

OldOp includes ->range, it is the same as the range of the OpDcl. Renaming is accomplished by applying first the OpRenames, then the SortRenames.

ASSOCIATED THEORY

A theory is associated with each trait. A theory is an inference-closed set of well-formed formulas (wff) of typed first-order predicate calculus with equality. The familiar meanings of the equality symbol (=), the propositional connectives (&, ¦, =>, ...) , and the quantifiers ($\forall$ and $\exists$) are used. The traits Boolean and Equality give the operators for the propositional connectives and = the same meanings.

The theory associated with a simple trait is defined by:

* Axioms: Each equation, universally quantified by the VarDcls of its axioms is in the theory.

* Inequation: ~(true)=false is in the theory

* First order predicate calculus with equality: The theory contains the axioms of conventional first order predicate calculus with equality and is closed under its rules of inference.

* Induction: If the trait has a generators with SortId S and a bylist **by** $[op_1,...,op_n]$, and P(s) is a wff formula with free variable s, of Sort S, then the theory contains the wff

$$\forall[s : S]\ P(s)$$

if for each opi in $[op_1,..,op_n]$

$$Q_i \Rightarrow p(op_i(x_1,..,x_k))$$ is in the theory

where k is the arity of op_i,

the x_j's are variables not free in P, and
Q_i is the conjunction of $P(x_j)$ for each j such that the j-th argument of op_i is of sort S.

* Reduction: If the trait has a partitions with SortId S and a bylist **by** $[op_1,...,op_n]$, the theory contains the wff

$$\forall[s_1, s_2 : S](Q \Rightarrow s_1=s_2)$$

where Q is the conjunction, for each op_i, and each j such that the j-th argument of op_i is of sort D of

$$\forall[x_1:S_1,...,x_k:S_k] \quad (Subs(op_i,j,s_1)=Subs(op_i,j,s_2))$$

where $S_1,..,S_k$ is the domain of op_i, and
Subs(op,j,s) is $op(x_1,...,x_k)$ with s substituted for x_j.

IMPLICIT INCORPORATION OF BOOLEAN, IFTHENELSE AND EQUALITY

The standard traits are implicitly incorporated as needed into other traits to assure uniform meanings for the operators they constrain.

```
boolean : trait
     introduces
          true :      -> bool
          false:      -> bool
          ~ #   : bool -> bool
          # & #: bool,bool -> bool
          # ¦ #: bool,bool -> bool
          # =># : bool,bool -> bool
          # <=> #: bool,bool -> bool
     asserts bool generated by [true,false]
     for all [b :bool]
          ~true = false
          ~false = true
          (true & b) = b
          (false & b) = false
          (true ¦ b) = true
          (false ¦ b) = b
          (true => b) = b
          (false => b) = true
          (true <=> b) = b
          (false <=> b) = ~b
     implies converts [~, &, ¦, =>, <=>]
```

```
equality : trait
     introduces
          # = #: T,T      -> bool
     asserts T partitioned by [ = ]
     for all [x,y,z : T]
          (x=x)
          (x=y) <=> (y=x)

ifThenElse : trait
     introduces IfThenElse : bool, T, T -> T
     asserts for all [ t1, t2 : T ]
          ifThenElse(true,t1,t2) = t1
          ifThenElse(false,t1,t2) = t2
     implies converts [ ifthenelse ]
```

SEMANTIC CHECKING

Each trait must be logically consistent, discharge the assumptions of its external traits, be a conservative extension of its imports, be properly constraining, and imply its consequences.

Consistency:

A traitbody is consistent if the associated theory does not contain the equation true=false.

Assumptions:

Let A(T) be all the assumes of the traits imported or included in T, and R(T) be the result of translating T after removing these assumes. A(T) is discharged by T if the theory associated with the translation of each traitRef of A(T) is a subset of the theory associated with R(T).

Imports:

If trait T1 imports trait T2 and W is a wff containing only operators introduced in T2, W is in the theory associated with T1 if and only if it is in the theory associated with T2.

Constraints:

A propPart is properly constraining if it implies properties of only the operators in its constrains. The occurrence of a SortId in a constrains stands for the list of all SortedOps in the containing traits opPart whose signatures include that SortId.

Let T be a trait, and P the propPart

constrains SortedOp*, **so that** props

P is properly constraining in the trait consisting of T plus P if and only if each wff in the theory associated with T plus P is also in the theory associated with T or else contains a sortedOp listed in SortedOp*.

Consequences:

A trait implies its consequences if two conditions are met:

ConseqProps:

The theory associated with ConseqProps must be a subset of the theory of the trait in which the consequences appears. The theory asssociated the traitBody

includes traitRef*,

opPart

asserts props

where traitRef*, and props form the conseqprops, and opPart is the opPart in which the consequences appear.

Conversion:

Let C be a conversion. For each term, t, that contains no variables of any sort appearing in a generators in the containing trait, the theory of the containing trait must either

contain an equation t = t1, where t1 contains no SortedOp appearing in C's SortedOp*, or

contain an equation t' = t1, where t' is a subterm of t, and t1 is an instantiation of a term appearing in an exempts of the containing trait.

APPENDIX B : EXAMPLES

The input and output files in this appendix provide examples of the various semantic checks.

1 PARSING AND GENERATING REDUCTION RULES

Inverse.lch:

```
inverse : trait
introduces
     # - : sym -> sym
     # * # : sym, sym -> sym
asserts for all [x, y : sym ]
     ( x - ) * ( x * y ) = y
```

Inverse.dmp:

```
RELATIONSHIPS

ELEMENTS

        Sort
-------------------------------------------------------------

          sym
           :

    Operator Weight       Range      Domains
-------------------------------------------------------------

           *      2          sym        sym          sym
           -      1          sym        sym
```

Variable	Sort	Trait
x	sym	inverse
y	sym	inverse

THEORY

Axioms

1 : (x -) * (x * y) = y

Reduction rules

1 : (x -) * (x * y) -> y
2 : ((x -) -) * y -> x * y
3 : x * ((x -) * y) -> y

2 CHECKING ASSUMES

Top.lch:

```
top : trait
includes top1
```

Top1.lch:

```
top1 : trait
assumes top2
```

Top2.lch:

```
top2 : trait
introduces
   top2 : T -> T
asserts for all [top : T]
   top2(top) = top
```

Top.dmp:

```
RELATIONSHIPS

Level 0:
     top includes: top1
Level 1:
     top1 assumes: top2

ELEMENTS
        Sort
```

	T			
Operator	Weight	Range	Domains	
top2	1	T	T	T

Variable	Sort	Trait
top	T	top2

THEORY

Assumptions

1 : top2(top) = top

Reduction rules

top.err:

top.LCH (l 2, c 14) - error (51) Assumption not discharged
top2(top) = top

1 Error found

3 CHECKING IMPORTS

itest.lch

```
itest : trait
imports itest2
introduces
     itestop : I -> I
asserts for all [i : I]
     itestop(i) = i
     itest2op(i) = i
```

itest2.lch

```
itest2 : trait
introduces
     itest2op : I -> I
```

itest.dmp

```
LARCH Translator Specification:   itest.LCH

RELATIONSHIPS

Level 0
      itest imports: itest2

ELEMENTS

        Sort
_____________________________________________________

           I

    Operator Weight       Range      Domains
_____________________________________________________
```

```
    itest2op      1            I            I
     itestop      2            I            I

    Variable          Sort      Trait
-------------------------------------------------------------------------

           i             I        itest

THEORY

Axioms
-------------------------------------------------------------------------

  1 :  itestop( i ) =  i
  2 :  itest2op( i ) =  i

Imports
-------------------------------------------------------------------------

  1 :  itest2op( i ) =  i

Reduction rules
-------------------------------------------------------------------------

  1 :  itestop( i ) ->  i
```

itest.err

```
LARCH Translator Specification:   itest.LCH

itest.LCH (l 7, c 11) - error (52) Not a conservative
extension of imported trait   itest2op(i) = i
1 Errors found
```

4 CHECKING IMPLIES

Central.lch

```
central : trait
introduces
     # * #      : sym, sym      -> sym
asserts for all [ x, y, z : sym]
     (x * y) * (y * z) = y
implies asserts for all [ ]
     x * (( x * y) * z) = x * y
```

Central.dmp

LARCH Translator Specification: central.LCH

RELATIONSHIPS

ELEMENTS

Sort

sym

Operator	Weight	Range	Domains	
*	1	sym	sym	sym

Variable	Sort	Trait
x	sym	central
y	sym	central
z	sym	central

THEORY

Axioms

---

1 : (x * y) * (y * z) = y

Consequences

1: x * ((x * y) * z) = x * y

Reduction rules

---

1 : (x * y) * (y * z) -> y

2 : y * ((y * z) * z1) -> y * z

3 : (x1 * (x * y)) * y -> x * y

central.err

LARCH Translator Specification: central.LCH

APPENDIX C : LARCH AND PROLOG SYNTHESIS FROM REDUCTION RULES

Prolog is a logic programming language with built in unification and resolution. Programs in Prolog closely resemble Larch specifications. For these reasons it's potential as a language for implementing Larch was investigated. TURBO Prolog was used in the examples below.

Larch specifications can be easily translated into Prolog and the resulting Prolog program executed to check the semantics of the Larch specification. This same program can then be used to answer questions about the specification.

Translating a Larch Specification into Prolog

Take a simple specification for a stack as an example.

```
stack : trait
introduces
        new       :          -> stk
        isnew     : stk      -> bool
        pop       : stk      -> stk
        top       : stk      -> el
        push      : stk,el   -> stk
asserts stk generated by [new, push]
        stk partitioned by [top, pop, isnew ]
so that for all [s :stk, e: el]
```

```
                isnew(new)
                ~isnew(push(s,e))
                top(push(s,e)) = e
                pop(push(s,e)) = s
        implies converts [isnew]
        exempts top(new), pop(new)
```

A first-order theory is associated with the specification. The theory for stack would include the axioms, inequation, implicit incorporation of boolean, induction and reduction. This theory can be described in Prolog clauses, which can be mechanically determined from the specification, as shown below (which includes clauses for semantic checking).

```
    nowarnings

    INCLUDE "boolean.pro"

    domains
        el = symbol
        stk = new;
              push(stk,el)
    predicates
        isnew(stk,bool)
        top(stk,el)
        pop(stk,stk)
    clauses
        isnew(new,true).
        isnew(push(_,_),false).
        top(push(_,E),E).
        top(new,_) :-
                write("error - top(new) exempted\n"), fail.
```

```
        pop(push(S,_),S).
        pop(new,_) :-
                write("error - pop(new) exempted\n"), fail.

    /* equality.pro included with el for T, stk for T */

    predicates
        o_el_equal(el,el,s_bool)
    clauses
        o_el_equal(V_x,V_x,c_true).
        o_el_equal(V_x,V_y,c_false):-
            not(o_el_equal(V_x,V_y,c_true)).

    predicates
        o_stk_equal(s_stk,s_stk,s_bool)
    clauses
        o_stk_equal(V_x,V_x,c_true).
        o_stk_equal(V_x,V_y,c_false):-
            not(o_stk_equal(V_x,V_y,c_true)).

    /* partition clauses  */

    predicates
        o_partition(s_stk,s_stk,s_bool)
    clauses
        o_partition(c_new,c_new,c_true).
        o_partition(c_new,_,c_false).
        o_partition(_,c_new,c_false).
        o_partition(V_s1,V_s2,c_true) :-
            o_top(V_s1,V1), o_top(V_s2,V2),
            o_el_equal(V1,V2,c_true),
            o_pop(V_s1,V3), o_pop(V_s2,V4),
```

```
            o_partition(V3,V4,c_true),
            o_isnew(V_s1,V5), o_isnew(V_s2,V6),
            o_bool_equal(V5,V6,c_true).
      o_partition(V_s1,V_s2,c_false) :-
            not(o_partition(V_s1,V_s2,c_true)).

/* semantic checking */

predicates
      semantic_check
      consist_check
      convert_check
clauses

      semantic_check :-
            consist_check,
            !,   /* to avoid multiple solutions */
            convert_check.
      consist_check :-
            not(o_bool_equal(c_true,c_false,c_true)).
      consist_check :-
            write("failed consistency checking,
      true=false"),  fail.
      convert_check :-
            o_isnew(c_new,_),
            !,
            o_isnew(c_push(_,_),_).
      convert_check :-
            write("failed semantic check - isnew is not
            converted"),
            fail.
```

(Note - TURBO Prolog is typed, and has various restrictions, eg variable names must begin with uppercase letters, all of which has been taken into account. The o_ prefixes indicate an observer operator, c_ indicates a constructor. These are required to ensure conflicts between the specification and Prolog itself don't arise, eg c_true).

The Prolog code is executable, and questions about the trait can be asked (remembering to translate them into Prolog first).

More complicated examples

Stack could be defined in terms of other traits - as it has been done in the Larch handbook. Methods for synthesizing a Prolog program for this case were developed, and for generating the substantially more complex semantic checking.

Reasons for not using Prolog

There were numerous reasons for not using Prolog, including:

* Prolog itself is not complete, since it does not do an occur check.
* In order to check even a simple example, the specification will need to be translated, and the resulting program compiled and executed. A user interface would have to be built to translate Larch into Prolog clauses and to translate Prolog output into user readable form.
* Most of Larch's power lies in building up traits from

existing traits using "assumes", "imports" and "includes". Incorporating these traits and the resultant semantic checking becomes very complicated.

* It may happen that Prolog gets hung up on a question of the "too hard" variety, or, because of the way the user has recursively defined operations, the Prolog translation may wander off into a loop. Some external procedures would need to identify and interrupt these situations. An example is the standard Larch definition of equality.
* Prolog is mainly useful for checking a correct specification. If the Larch specification is incorrect or recursive as above, Prolog may become unpredictable and produce no solutions, many solutions or strange solutions. Prolog predicates for generating meaningful error messages may be difficult.

Are Prolog translations useful at all?

Possible uses of Prolog translations of Larch specifications are:

* Checking the functionality of the semantic checker.
* It is possible to generate an implementation of a specification in Prolog, using the Prolog clauses as a base. The user may have to give clauses for the constructor operations.
* An algorithm exists for generating test sets for implementations of abstract specifications along Larch lines. This algorithm uses Prolog and requires that the specification be translated into Prolog clauses.

Bibliography

TURBO Prolog, Borland 1987.

'Review of Turbo Prolog', Dr Dobbs Journal of Software Tools, September 1986.