Considerations on genericity for programming language design

Holger Gast

WSI 99-5

23. März 1999

Wilhelm-Schickard-Institut für Informatik
Arbeitsbereich Computeralgebra
Prof. Dr. Rüdiger Loos
Sand 13
D-72076 Tübingen

e-mail: gast@informatik.uni-tuebingen.de

ISSN 0946-3852

Abstract

The generic programming paradigm has received considerable attention since the publication of the C^{++} STL library [20].

The specification of capabilities, which a programming language must provide to support generic programming, has been examined in detail in [26] and [31]. Especially in the area of computer algebra, these requirements are extensive, because of the structural complexity of the types and algorithms and the need for precision in declarations.

In this essay we propose a general approach to the design and implementation of a type system. We then discuss how programming language constructs can be described by means of the calculus, including overload resolution, higher-order functions and type constructors with contravariant argument positions together with automatic instantiation of generic algorithms with bounded type variables (type classes).

Contents

1 Introduction **1**

2 Modeling in Programming Languages **2**
2.1 Why Model Reality ? 2
2.2 Approaches along/through Programming Paradigms 2
2.2.1 Imperative Programming 3
2.2.2 OOP 3
2.2.3 Term Algebras 5
2.2.4 Order Sorted Algebra 6
2.2.5 Functional/Applicative Programming 7
2.2.6 Logic Programming 9
2.2.7 Tecton 9
2.3 Problematic Modeling Situations in Different Approaches . . 10
2.3.1 Records of Functions 11
2.3.2 OOP: Circle-Ellipse Dilemma 12
2.4 Dependent Types 13

3 Preliminary Statement of Goals **14**
3.1 Aspect: Emphasis on Orders 14
3.2 What Parameters can an Algorithm have ? 16
3.3 Functions Returning Types 17
3.4 Overload Resolution 17
3.5 Hierarchy of Objects 18
3.6 Partial Evaluation Strategies 19
3.7 No Implicit Statements 19
3.8 Explicit Version of Co-/Contravariance 19

4 On Languages and Reduction **20**
4.1 Languages 21
4.2 Reduction 22

5 A Unifying Approach **23**
5.1 On Well-Typing 24
5.2 Hierarchical Objects 25
5.3 Typed Execution 27
5.4 Decision Language : Inference-trees 31
5.4.1 Eliminating the [forget] rule 32
5.4.2 Linearizing Proofs 33

5.4.3 Incremental Computation of Substitutions 35
5.4.4 Non-deterministic Pathfinding 38
5.4.5 Generalization to Conditional Rules 39
5.4.6 Selecting the Best Proof 40
5.5 Language Variations/Extensions 41
5.5.1 Run-time Checks 41
5.5.2 Binary Predicates 43
5.6 The Problem of Error-Messages 43

6 The Otter Proof System **43**

7 Translation of Programming Language Constructs **46**
7.1 Notation for L_{TE} 47
7.2 Function Calls and Overload Resolution 48
7.3 Subsorts 53
7.4 Conversion 55
7.5 Higher-order functions 57
7.6 Object-Oriented Programming 61
7.7 Integrating Structures 62
7.8 Data Construction and Destruction 63
7.9 Calling and Computing with Generic Functions 65
7.10 Assignment Operations 67
7.11 Sequencing 68
7.12 Directing the Interpretation 68
7.13 User Definable Types 68

8 Extensions and Further Thoughts **69**
8.1 Towards Type Inference 69
8.2 Solving a Longstanding Problem 70
8.3 The Program Structure 71
8.3.1 Compiling Generic Functions 72
8.3.2 A Module System 73
8.4 Run-time Well-Typing 74
8.5 Symbolic Attributes as Restrictions 74
8.6 Verification Issues 76

9 Conclusion **78**

10 Acknowledgments **79**

1 Introduction

The paradigm of generic programming, whatever the exact details of a particular definition may be, tends to exploit programming languages to their fringe, if it is supported at all.

Therefore, it seems useful to investigate into the question, which specific features of a language are needed and to what extend for this style of programming. Especially it is desirable to put these single features into orthogonal categories and describe them independently from one another: In this way it will be possible to change little decisions in one part without affecting too much the rest of the system.

In principle, generic programming can be easily done in dynamically typed languages, such as SCHEME, but the arising (almost) total freedom and lack of security can scarcely be called "support."

The main idea and intention of this essay can be pinned down in one sentence:

> The better we can express *within* the programming language the ideas which are *behind* the scenes in writing programs, the better the compiler will be able to assist and check our reasoning.

Basically, there are two parts we need to speak about in our programs:

1. The computational component (algorithm), incorporating our procedure to solve the given problem.

2. The objects which our algorithms and data structures will be dealing with.

Here, we not at all speak about the best choice for the first aspect, but the latter one is in the focus of our interest. Since the aspect of describing objects is part of the type system of a language, this essay can be seen as related to questions of type theory. However, it is strongly heading towards practical applications, not "beautiful" results.

In section 2 we analyze the devices of several outstanding languages, in order to motivate the derivations later on, from a particular point of view. Our goal is to have an explicit collection of problems to be solved, which can be found in the intentions of the particular languages, i.e. the reasons why they were invented.

What the final aim is, will become clear in section 3, where we abstract from any specific language and try to capture the essence of their capabilities of modeling.

Building on this, section 5 will contain our system, with the core in section 5.3, which is related towards to the well known programming language features in section 7, where we give a translation for each feature separately into our calculus, thereby showing that the other languages are no more expressive with respect to the points under consideration.

The motivation for developing the system presented in this essay stems from the author's work on the SUCH THAT language proposed by Sibylle Schupp [26], the implementation of which was done in a prototypical form in [14, 9]. The experiences in this project indicated that a more general approach was necessary for the decision procedures used in the compiler.

2 Modeling in Programming Languages

In this section we want to briefly motivate the strife for expressive programming languages, not that much because we feel that was necessary, but rather to collect efforts already undertaken and gather their goals and crucial ideas in one place.

2.1 Why Model Reality ?

Historically, the von Neumann architecture is the basis for modern computing and the assembler/machine code used in this machine model has influenced a great part of imperative programming style. With the arising of larger pieces of software, most notably libraries, the need to communicate the *usage* of software among programmers in terms of *interfaces* became obvious.

If the *intention* of a library-designer can be expressed in the programming language, then the correct usage of the library can be (partly) *ensured* by the compiler, leading to more reliable programs and therefore better software-development.

Furthermore, if the possible expressions in the programming language model some sort of reality (e.g. mathematical notions), which fits well the purpose of the program being written, then *no translation* on the programmer's part is necessary when reading or writing interfaces.

2.2 Approaches along/through Programming Paradigms

Different approaches have been taken to make these interface descriptions both readable and checkable by a compiler for the programming language. At the same time, the level of abstraction has been increased step-wise

to allow for writing flexible and generally useful software and freeing the programmer from the burden of thinking "at machine level."

2.2.1 Imperative Programming

Extending the notion of the assembler-level manipulation of registers and memory directly naturally yields what has been called the imperative languages, where memory is directly visible and can be therefore manipulated by the programmer.

Although it has been argued that the style of programming is out-dated and low-level, we would still like to include it into the discussion in the present essay. As Milner [18, p.373] observes in his conclusions:

> What is rather needed is a language design which pays more respect to side effects; [...]

Furthermore, constructs such as looping and invariants have a semantics in their own right and are used extensively in the community of algorithms-research to give precise and elegant descriptions of procedures [6], [28], [23] together with the rigorous analysis of computational bounds and correctness of algorithms.

We summarize the discussion in the following features:

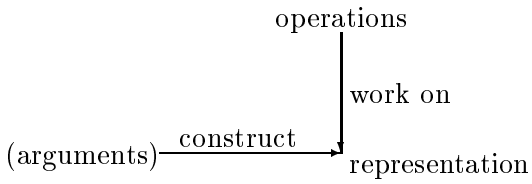
- assignment
- variables & parameters declared
- explicit representation
- transparent workings
- machine model near physical computer

2.2.2 OOP

Object-oriented programming originally started out with languages such as Simula and SmallTalk in the field of physical simulation and with the intuition, that once we describe precisely the properties and behaviour of a single object and define interactions among them, then we will be able to create larger system from these building-blocks. This approach naturally extends to any area, which is dominated by *self-contained*, largely independent objects, which have an *a-priori meaning* by themselves, most notably window-systems for graphical user interfaces (unlike e.g. the integers, which receive their algebraic structure only when combined with *other integers*).

The combination of both data items and operations on them into objects tries to model notions of the real world in programs. It inherits the idea of memory modification from imperative languages but partly hides the details in the abstraction of member-variables, whose exact address, for example, are not known to the programmer and also the representation of the objects themselves can be abstracted from in favour of a general object model, describing the essential behavioural properties only.

Concerning the representation of an object's data, we observe that in usual languages constructors of objects and the operations, i.e. methods, have the following relation:



SmallTalk is *typed only at run-time*, i.e. a message sent to an object can "go wrong" if the object is not able to process it. In the late 80's the object-oriented paradigm has received a considerable attention which again invoked a discussion about solid theoretical foundations.

A widely accepted model was born:

- Objects are records of data and functions.
- Objects can have a subtype-relation $<:$, which models (the semantical notion of) inheritance and contains the intuitive idea that if message m can be sent to object o_2 and $o_1 <: o_2$, then m can also be sent to o_1.

The details, however, were still more involved than had perceivably been expected from the experiences with SmallTalk. This will be discussed in the next section.

As in this essay we are mostly concerned with modeling, we want to strongly emphasize at this point the relation of inheritance which, again intuitively, has been derived from the observation that some objects are *special cases* of other objects and that we want to be able to express this *relation* within the programming language.

In connection with the CLU language (which is strictly speaking not object-oriented) B. Liskov according to [32] seems to have introduced the similar idea of *substitutability*, saying that in any situation a CLU cluster

A may be taken in place of a cluster B iff A is substitutable for B. Because of the resemblance, inheritance is often taken to express substitutability.

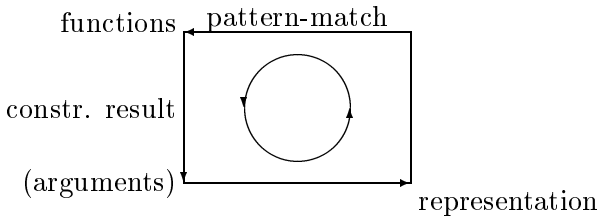
- (localized) assignment of member variables
- encapsulation
- combination of data and operations
- representation of data chosen by programmer (constructors compile to internal representation)
- subtyping/inheritance = isA-relation

2.2.3 Term Algebras

Universal algebra incorporates the notion of carrier sets, whose exact structure is neither known nor important to express properties about the structures under consideration.

To obtain an executable algorithm[1], a model of the universal structure must be found and the term-algebra, which is build inductively over (value-) constructors for the different sorts, serve well this purpose and impose a tree/dag-like structure on *any* data item, which can be represented. It follows, that once a member of a sort has been constructed, it can be analyzed to yield the constructor and its arguments again, given that appropriate information has been stored.

On the negative side we remark that standardized construction inhibits a specialized representation (e.g. compression), i.e. the diagram from OOP now looks like this:



Therefore construction and pattern-matching are the essentially necessary and sufficient operations on *any* value, which evolves into a uniform description within programs.

[1] Distinguished from a generic algorithm schema in this context !

- standardized construction and pattern-matching
- structural induction
- no information on representation
- sorts carry semantics

2.2.4 Order Sorted Algebra

Order sorted algebra adopts the term-algebra model of data but extends it in two directions: First, rewriting techniques most naturally fit into the picture as a model of computation (which yields a theory of many-sorted algebra for computation). Second, it is observed that often in programming practice, we need to talk about *sub*sets of sets, which translates to subsorts, which are assigned a *restricted repertory* of value-constructors.

Apart from the elegant description of certain situations, one of the motivations for describing programs in terms of equality of expressions (i.e. rewriting equations), is the aim of *verifiable programs*, which is driven further in the TECTON[19] language. The situation of library designers and users, which has been sketched in the introductory passage, is addressed again in the most rigorous interpretation, i.e. the equivalence of systems of *first-order formulae*.

Goguen and Meseguer have been pursuing the development of OBJ[10], [11] as a realization of these ideas and also the functional programming paradigm, which will be discussed next, is connected interestingly [22]. OBJ also provides an advanced system for grouping carrier-sets and functions to allow to build structures and transform them into (*view as*) one another.

Parallel to the Haskell's type classes, OBJ has *theories*, which can be instantiated with different realizations, called *objects* and then subsequently used to program further functions.

- relation $<$ on sorts: set-theoretic inclusion, subset of value-constructors allowed
- matching and rewriting as computation
- module systems for grouping functions and carrier-sets to algebraic data types

2.2.5 Functional/Applicative Programming

Superficially, what constitutes the difference between many-sorted algebra and functional programming, is the model of computation. Starting out with the λ-calculus, and extending it with term algebras as representation of data (which is more convenient than pure λ-terms in normal form (such as the Church numerals)), the approach yields programming languages which behave along the lines of mathematical intuition.

A second characteristic feature, which is also directly contained in the λ-calculus is the possibility of using higher-order functions, i.e. functions, whose arguments can again be functions.

Programs here, in the pure form, consist of expressions only, together with functional abstraction over one variable. What is totally invisible is the representation of terms and functions, the underlying memory structure and in some cases even the *order of evaluation*. Indeed, lazy-evaluation, which consists of the two parts

1. no expression is evaluated unless absolutely necessary
2. no expression is evaluated more than once

is modeled after the mathematical model of terms: These stand for some value, which is most of the time not interesting, but could be computed[2] if needed.

Because of the inductive structure of the calculus, it is possible to design comprehensive type-systems, which provably ensure run-time type correctness. We remind the reader of the series of languages starting out with ML [18] and continuing with Haskell [30], [12], [21] and Gofer [16], [15]. While ML introduced the idea of type-inference for types built inductively of type-variables, simple types and the type-constructors $\rightarrow$, $\times$, [],... Unlike in OSA, where sorts do not carry any structure besides that induced by the value constructors, we find here that *types are terms* again.

With the presented algorithm $\mathcal{W}$ [18] it is possible to assign a type to (correct) expressions *without* requiring declarations of variables, etc. by the programmer. ML has *unbounded* type variables, i.e. it is in particular not possible to introduce constraints on the instantiation.

Haskell extends the system by *type classes*, which describe sets of types which have to provide operations (recall the universal algebra semantics with carrier set and operations). Jones has shown in [16], [15] that an even more general approach is possible for languages with type inference. Haskell's

[2] assuming that the required functions are effectively realized

original system [30] involves type *contexts*, i.e. constraints on type variables in the form of membership of a specific type class, which is written as

$$\texttt{C}\ a \Rightarrow \sigma(a)$$

where $\sigma(a)$ is a type involving the type variable. Jones identifies a class `C` in the context with a predicate π to be true for a substitution for the variable and writes the above type as

$$\{\, \sigma(a) \mid a \text{ is a type such that } \pi(a) \text{ holds} \,\}.$$

Inference rules about the predicates, as derived from `class` declarations, are then given in the form

$$\{\pi_1, \ldots, \pi_n\} \Vdash \{\pi_{n+1}, \ldots, \pi_m\}.$$

$\Vdash$ is also used to denote the transitive closure of the corresponding assumptions.

Accordingly, a type judgment consist of 4 parts:

$$P \mid A \vdash x : \sigma$$

- A set of predicates which is currently valid for the type variables.
- An assignment of types to variables.
- The term to be typed.
- A type for the term.

Within this framework, a syntax-directed scheme for type inference can be given, which has properties similar to Milner's algorithm $\mathcal{W}$.

The whole work is however aimed towards a non ad-hoc *overload resolution* and the implementation details [21], which make use of higher-order functions and dictionary lookup at run-time, seem to incur a serious inefficiency when contrasted with the careful design of C^{++} to decide most things at compile time.[3]

It seems worthwhile to note that Standard ML [13] also incorporates an extensive module system, and instantiations of modules can be roughly compared to again building structures of sets and operations, which are fully exploited only in Haskell's type classes.

[3] Of course, partial specialization techniques can be used in this context if desired.

- simple structure ⇒ proofs for program correctness
- no assignment
- rich theory: machine models, mathematical semantics
- extensive type-theory: type inference
- lazy-evaluation / order not predictable
- types are terms, built from constants, variables and constructors ($\rightarrow$, $\times$, []).

2.2.6 Logic Programming

First-order logic formulae cannot only be used for a meta-description of programs to allow for verification, but also as a programming language itself. Computation is then interpreted as a search for a proof, although the Prolog *depth*-first search strategy with back-tracking and a cut-rule is not complete in the strict sense of logic.

However, it seems still interesting for the later discussion of the OTTER system in our context, to note that in principle first-order systems with equality allow to express rewrite-rules and are therefore Turing-complete, provided that the search procedure is complete, i.e. it finds a proof for every provable formula[4].

Following the predicate calculus (see [8]) the objects which the languages talk about, are again *terms* and the *predicates* which describe properties of terms, are the computational part. It follows that pattern matching and term-algebras are again crucial concepts.

- pattern matching/unification & terms
- logical inference as computation

2.2.7 Tecton

At a superficial glance, a TECTON[19] program can well be confused with programs written in OBJ, because again we can talk about (sub-)sorts, declare functions and give requirements for these. However, there are two major and crucial differences, which make TECTON unique among the languages in existence:

[4]Note the slight subtlety of "completeness" notions: In [8, p. 133] a theorem states that a refutation is found by resolution *if* the set of clauses is unsatisfiable. For satisfiable systems (i.e. the initial judgment is false) the procedure may well run forever).

- The requirements are not necessarily constructive rules for reduction and computation, but really describe (in full first-order logic) the *minimal* expected properties.

- The *intention* is to group together requirements and thereby reuse proofs, which can be grouped in a similar fashion.

  This is achieved by using already declared structures, here called *concepts*, in new concepts as *parts* and *superiors*. If A is a part of B then A is available for substitution by any concept, which fulfills all the requirements of A. In both cases, B can be treated as A by definition of the language semantics. It follows that the question, whether A' fulfills the requirements for A can often be decided without looking at the requirements, simply because A is a superior/part of A.

  This approach can help minimize the impact on compiler-efficiency which would be noticed if theorem-proving in first-order logic was the only basis for decisions and thus open the door to designing larger software systems with meticulous compiler support.

Such a cascading description of requirements also suits well the idea of generic programming, which we do not want to define here, but which has the intention of defining an algorithm in terms of the minimal requirements of its parameters.

The decision to make full first-order logic available for the description, encompasses the possibility to write totally correct programs, which can be instantiated and thus reused arbitrarily by substitution of parts and the resulting instances will again be correct.

- sorts with set-inclusion $<$
- not necessarily constructive description with full first-order logic
- grouping of requirements and proof-obligation
- obliviate full theorem proving by sufficient syntactic conditions

2.3 Problematic Modeling Situations in Different Approaches

Whenever a language is designed to serve and model well a particular area of application, it can be expected that it has drawbacks in other respects. Just as the last section was not supposed to describe precisely the achievements

of single approaches and the reasoning behind them, listing difficulties now is not meant to provide a reason to condemn the languages, rather do we again want to gather previous insights.

2.3.1 Records of Functions

We have described above that OOP has been derived from the wish to model reality and in particular the observation, that some objects are special cases of others.

In connection with the theoretical treatment of inheritance several problems have occurred and have been formalized by Cardelli [3], together with Abadi [1], Castagna [4] and others. In the development of EIFFEL and later SATHER the most imminent question has been discovered and practically described; given two records of functions and data as the following

$$\begin{array}{lll} o1 = \{ & & o2 = \{ \\ \quad x & : & t_{x1} \qquad\qquad x & : & t_{x2} \\ \quad f & : & t_{d1} \to t_{c1} \qquad\qquad f & : & t_{d2} \to t_{c2} \\ \} & & \} \end{array}$$

What are the restrictions on the occurring t_X to perceive $o2$ as having inherited from $o1$ (for short $o2 <: o1$) and overridden some of the members, still in the sense that any message legal for $o1$ can also be sent to $o2$?

Essentially the following cases arise, depending on what manipulations we allow to be performed on objects. We take the relation $<$ to include, somewhat abusing notation, both $<:$ and usual convertibility of built-in types. As a justification, the two meanings cannot be confused, because one applies when sending messages and the other when we execute a built-in function.

reading x If a function may read a member variable and is expected to work correctly afterwards, then certainly $t_{x2} < t_{x1}$ for $o2 < o1$ (the value of x_2 must be substitutable for x_1).

writing x If a function (outside of the objects) is allowed to write to x, then we need $t_{x2} > t_{x1}$, i.e. t_{x2} is less specialized. The reason is obvious: In order to get substitutability of $o2$ for $o1$ any value assignable to x_1 must be assignable to x_2, i.e. x_2 may not require more than x_1.

Invoking f By the same reasoning as before, for the codomains we need $t_{c2} < t_{c1}$. Since calls to f are coded independent from whether the f of $o1$ or $o2$ is invoked, f_2 must accept at least the parameters acceptable for $o1$, i.e. we need $t_{d2} > t_{d1}$ (contrary to the codomain !).

This characteristic behaviour, known as the co-/contravariance problem of inheritance [4], has the consequence that a derived (=more specialized) object may *not* override a method with a more specialized parameter list, which is often needed, as soon as objects are not isolated entities [32].

One important special case of this situation shows up when a method of an abstract base class (for instance `Comparable`) has a parameter of that class (`compare_to(Comparable y)`). Cardelli investigates into this problem of `self`-specialization. in [1, p.23] and finds solutions for example in recursive record types (the μ operator) or by making `self` a special type name with the desired properties by definition.[5]

Another possibility is to introduce $\exists$ to bind type variables. The expressions $\exists\alpha < t_1.X$ and $\exists\beta < t_2.Y$ show the desired property with

$$(\exists\alpha < t_1.X) < (\exists\alpha < t_2.Y) \iff (t_1 < t_2 \wedge X[t_1'/\alpha] < Y[t_2'/\beta)$$

for all $t_1' < t_1$ and $t_2' < t_2$. However, then severe restrictions have to be placed on the return value [1, p. 173].

In SmallTalk, where typing was only dynamic, these problems did not arise until run-time and here the special knowledge of the programmer was needed to avoid conflicts. Clearly however, the substitution principle is in danger, once we replace a $\forall$ by "special knowledge", which corresponds to invariants on the object interactions.

2.3.2 OOP: Circle-Ellipse Dilemma

The preceding paragraphs have collected some theoretical problems, which *follow* from the OOP approach if we want to introduce static type safety. However, there are also well-known problems on the modeling part, i.e. situations in which one wants to use inheritance for some reasons and yet cannot assert substitutability for others.

Weihe [32] observes, that although apparently circles and ellipses have a certain relation, we cannot easily model this relation by inheritance. Surely, an ellipse is not a circle and also the converse is not true, if ellipses include a function `stretchXY` which modifies the radii independently.

For an explanation with respect to the current context, we suggest the following reasoning: With a class, there are certain implicit invariants[6] and with respect to these the operation `stretchXY` is not closed when applied to circles. Writing the functions explicitly, they *should* be declared as:

[5] Both the Tecton and the OSA approach do not incur these problems, because the carrier sets are explicit in the sorts, and not implicit in the notion of "this object."

[6] In Eiffel/Sather we can make them explicit.

for ellipses: ellipse×real×real → ellipse
for circles: circle ×real×real → ellipse

With this, we indeed *can* stretch circles, only we can't expect to get a circle, but an ellipse !

Object-oriented languages fail to express this declaration, because the first parameter is implicit in the receiver of the message (of type `self`) and the last return value again is implicit in the modification of member variables. Because of the assumed `self`-specialization (see [1, p. 23]) in inheritance, the return value for circles is inevitably `circle`, which has to be avoided.

Recall however, that the `self`-type has been introduced to allow for specialization of parameters in method-overriding. If we don't go that way, then it has been suggested to treat inheritance as conversions — and this yields the correct semantics: Coerce a circle to an ellipse, apply the ellipse-operation stretch to get — an ellipse.

To make a sharper point, the choice is the following: The conversion approach yields correct specialization behaviour for *return* parameters, the implicit `self`-specialization gives us a desirable treatment for the *argument* positions of methods.

2.4 Dependent Types

As long as the types of an object (now not in the OOP sense, but including variable locations and constants) can be fully determined statically, one could expect that it is only the complexity of the compiler which will increase with more sophisticated language designs, but essentially every question will be answered at compile time. However, this scope is quickly exceeded as soon as we try to model precisely situations taken from mathematics. Now the value, not only a static type, decides on the allowed operations on an object – and this value may not be known until the code is actually executed.

Immediately we find the following dilemma for a library designer when writing a generally useful type for $\mathbf{Z}/n\mathbf{Z}$. The question is, speaking in C^{++} terms, whether n should be a static template-value parameter, or a dynamic parameter to the value-constructor. The trade is between flexibility and efficiency:

```
template <int n> class Mod {
  Mod operator+(Mod &c) { ... };
  ...
};
```

```
class Mod {
 int n;
 Mod(int N) : n(N) { };
 ...
};
```

Worse than the probably small run-time penalty of the flexible version with dynamically determined n, is the restricted knowledge about n at compile time. Having it a prime number, the class can also provide a / operator which makes it an instance of a field, while in the general case, it is only a ring[7]. Indeed, C^{++} cannot even check automatically that the $n_{1/2}$ of the two parameters to the + operator must be the same. ADA can insert such run-time checks for dependent record types (whose type depends on the value of record fields) and even resolve, as an optimization, those checks at compile-time, which only involve constants.

3 Preliminary Statement of Goals

The previous sections have concentrated on gathering examples of hot spots in language design with respect to modeling capabilities. This section contains observations derived from or related to those, but at a more abstract level. The following remarks are neither intended nor appropriate to give answers to detailed questions, but may well serve as a general guideline for structuring a language.

3.1 Aspect: Emphasis on Orders

In the discussion in the literature, if a relation on types is introduced, it is usually a preorder, i.e. it is taken to be reflexive and transitive or the transitive and reflexive closure is examined. Whenever directed acyclic graphs can be used to represent the relation, which is true in OOP and OSA, these assumptions imply antisymmetry, which means we deal with partial orders. In OSA and functional programming [22] other desirable properties (such as uniqueness of overload-resolution) are induced by further requirements, e.g. that for any two sorts A,B there exists a C with $A < C \wedge B < C$, which leads to (upper) semi-lattices as the relation on types.

[7] Of course C^{++} itself does not, without using template specialization for every prime number in the range of `int`, provide such reasoning capabilities, but a new language should.

Cardelli [3] and Castagna [4] have introduced the terms *covariant* and *contravariant* for some expression X:

$$X \text{ covariant in } \alpha \iff X[s/\alpha] < X[t/\alpha] \text{ whenever } s < t$$

and

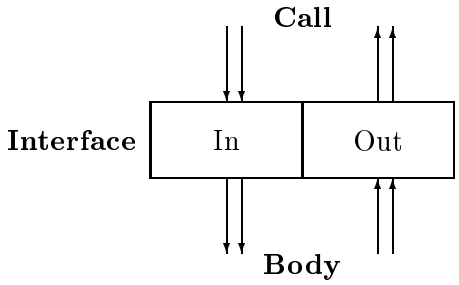
$$X \text{ contravariant in } \alpha \iff X[s/\alpha] < X[t/\alpha] \text{ whenever } t < s\,.$$

This definition is well-suited for examining expressions, because when combining subexpressions to an expression via a constructor, the resulting expression will again be co- or contravariant and Plümicke [22] suggests simply annotating each variable with $+$ or $-$ for this purpose (but only in the *internal* workings, not the language). Weber [31, p. 54] also goes into this direction and divides all parameters of a (type-) constructor into the sets $S_<, S_>, S_=$, which also allows the programmer to require *in*variance.

Note however, that the notion of co-/contravariance has been induced by the natural meanings of the $\forall$ and $\exists$ quantifiers and $\to$ constructor when we allow the user to include these into type-expressions. Also Reynolds [24] suggested to do so, with the Π and Σ product and sum types and Λ type-variable abstractions. The bounds of the quantifiers are stated as (for example) $\forall(\alpha < X) : Y(\alpha)$. Contravariance of this expression is *implicitly* connected with the $\forall$ operator.

We will come back to this idea in a moment.

As another principle, we want to be able to give type information for the interface only and then infer that the implementation, since it was written for this interface, does not go wrong, if only the restrictions of the interface are obeyed. Indeed, the *only* thing, which we can rely on in the description of the type system at this point, is *transitivity* of the order-relation on types:



We can exploit this observation by designing a type system, which *only* relies on transitivity of $<$ and does not have any other, often artificial rules

(as e.g. in the C^{++} standard). Then writing a correct program can be reduced to the task of avoiding the situation $A > B < C$, from which no relation between A and C can be inferred.

The properties of $<$ can be (almost) arbitrarily chosen and we think it would be worthwhile to investigate several type systems from this point of view. Whenever additional elements, such as built-in functions and parameter-passing / method-sending strategies get involved, some properties of $<$ may be induced indirectly – for example if built-in functions cannot convert values, then $<$ has to ensure that primitive data types are in the correct representation *before* calling a built-in function. Similar reasoning will be necessary for call-by-reference, call-by-value etc.

3.2 What Parameters can an Algorithm have ?

Aside from the previous discussion, Reynolds [24] describes, how the λ-calculus can be extended with another abstraction operator Λ for type-expressions, not values (=terms). Transferring this idea to imperative programming, gives us that generic procedures simply do have type parameters:

$$f(t_1, \ldots, t_l; x_1, \ldots, x_m; y_1, \ldots, y_n)$$

This is a procedure f with l type parameters, m value parameters and n return values.

Although most imperative languages (except C^{++}) do introduce type parameters at an outer level, for example classes (EIFFEL/SATHER) or modules (ADA), we argue that the above notation is better suited to generic programming and encourage statements of minimal requirements:

- We don't need to instantiate a whole package if we want one algorithm (which is useful for large generic libraries).
- Since the programmer needs to state requirements for every algorithm separately, he/she is more likely to analyze them in detail, especially because
- Seldom two algorithms will have exactly the same requirements.

The approach yields several features as simple extensions to well-known ideas:

- Instantiation of a generic algorithm corresponds to (compile-time or link-time) partial evaluation.

- Bounds on type parameters are truly types-of-types (if we allow the type parameters to have the form $t : c$ where c is a type class)

The most appealing consequence of this approach, however, is that the strong distinction between type parameters, which are according to Weihe [32] often implicitly associated with static checking, and value parameters, which are evaluated at run-time, is ultimately lifted. Speaking metaphorically, this paves one lane of the way to a generic language.

On the negative side, we ask the reader to recall how cumbersome generic programming can be in ADA, simply because all type parameters must be given explicitly. Therefore we argue for nesting the type classes within the description of the value parameters, have the compiler instantiate them automatically, and *then* generate this explicit form of representation, suitable for treatment by a simple substitution-scheme in the linker of our language.

3.3 Functions Returning Types

If we have now allowed types as parameters to algorithms, why shouldn't we allow algorithms to return types ?

Of course, this possibility allows for computation with types such as in while loops (!), but we expect it to be seldomly used or useful and therefore don't discuss it in detail.

One very significant example problem, the sequence-sorted sequence dilemma as described in section 8.2, needs support of such functions however.

3.4 Overload Resolution

As Loos and Schupp observe in [27] overloading at least of function identifiers is essential to generic programming.

Essentially overload resolution must therefore fulfill two tasks, according to the algorithm-parameter view presented before:

- Select a unique function to be called for the overloaded identifier in the parse tree.
- Give the instantiation of the function's type variables, if any.

The first point can be handled by unique function names, given in the declaration together with the overloaded identifier or generated automatically (name mangling). The second objective can be interpreted as computing an algorithm's type parameters from its value parameters (see sec. 3.2). For

ADA, Baker [2] has given a bottom-up algorithm, which accomplishes non-generic overload resolution. In [14] we showed that this algorithm extends naturally to the generic case, i.e. information on instantiation is propagated up the parse tree by applying the substitution, which unifies actual and formal parameters also to the output of an algorithm. Essentially, we again have a directed way of computation, and the algorithm can therefore be expressed in the relation $\rightarrow$ to be introduced, if we allow conditions on the application of a rule (i.e. we can recursively check the parameters' types).

Section 7.2 gives the details.

3.5 Hierarchy of Objects

We have already mentioned that the above approach makes use of types-of-types as bounds of type parameters. And indeed, this lifts yet another restriction when carried out in full consequence: Types describe sets of values, type-classes (see Haskell) describe sets of types. And we can write this down as a hierarchy:

$$\begin{array}{ll} \text{level 1} & \text{value} \\ & \in \\ \text{level 2} & \text{type} \\ & \in \\ \text{level 3} & \text{class} \end{array}$$

Whether it would be useful to extend this diagram further down by classes of classes is a question of appropriate examples to be solved. For the examples in section 7 we need to introduce a class *computation*, which contains fragments of programs (including expressions). This class is connected to *types*, because we want to give the type of the result and to *value*. If and only if a computation reduces to a value, it is considered finished and can be used as a argument to a function call, for example.

In a later section, we will make use of the accomplished abstraction to embed overload-resolution in the resulting decision system.

Note that with the introduction of levels, we have the possibility to dispose of the common names and abstract from them to get a more uniform system without the need for special cases. However, we must, by designing the environment appropriately, ensure that the system behaves as expected nevertheless.

3.6 Partial Evaluation Strategies

We have already mentioned the relation between static instantiation of generic procedures and partial evaluation. More abstractly speaking, the goal must be two-fold:

- Take as many decisions as possible at compile-time time.
- Generate code to ensure that the other requirements are fulfilled at run-time or throw a descriptive error-message (at least with a pointer to the source-line and stack-trace) otherwise.

With this goal achieved, the design dilemma for the $\mathbf{Z}/n\mathbf{Z}$ does not arise: We can write the flexible version, knowing that the compiler will infer the more specialized code for n constant.

The special case is efficient, the general case is possible.

3.7 No Implicit Statements

Again we remind the reader of the co-/contravariance problem: It arose, because the quantifiers were *implicitly* associated with the co-/contravariant behaviour.

As the ultimate design goal for a language we state • No declaration has more than one meaning. • Every (useful) statement can be expressed.

3.8 Explicit Version of Co-/Contravariance

When we left co- and contravariance issues in the area of orders, the question was not quite answered how the two issues are related.

Suppose we want to declare a type constructor t with two parameters:

$$t(\alpha^+ : A, \beta^- : B)$$

Then for two instantiations $[X_1/\alpha, Y_1/\beta]$ and $[X_2/\alpha, Y_2/\beta]$ the resulting type will be more specialized if X_1 is more specialized than X_2 or Y_2 is more specialized than Y_1, i.e. we can lift a specialization "through" the application of a constructor. Suppose we restrict ourselves to using $<$ (and $>$) for expressing the same information:

$$t(X_1, Y_1) < t(X_2, Y_2) \textbf{ where } X_1 < X_2 \wedge Y_1 > Y_2$$

This means that we can reduce the decision to two other decisions, which have the same structure as the original one. Because we can make the recursion to $<$ explicit in the conditions, the decision procedure for $<$ does not need to take into account the intricate properties of structural conversion.

A translation will be necessary for conveniently writing in the programs (see sec. 4) but it will be straightforward.

> Suppose we want to build an algorithm for deciding $<$. Once we write rules as above to implement contravariant parameter positions, a term from the left hand side of a judgment plays the role of the right hand side of a sub-judgment: $Y_2 < Y_1$. Because variables can now possibly be found *left* of $<$, it follows that matching[8] as a primitive operation for binding variables is *not* enough; we will need unification.

4 On Languages and Reduction

In this section we want to introduce a notion of languages for judgments, which differs in some subtleties from classical languages of mathematical logic; however, these differences will be useful once we employ them to define a programming language, simply because the restrictions are aimed in this direction:

- Languages are focused on single judgments in a context and the context cannot be expressed in the language itself. The rules $\supset$: left and $\supset$: right from the Gentzen system for first order logic in [8, p.187] shows this deficiency directly:

$$\frac{,\,,\Delta \to A,\Lambda \quad B,\,,\,,\Delta \to \Lambda}{,\,,A \supset B,\Delta \to \Lambda}\ \supset\text{: left}$$

$$\frac{A,\,,\ \to B,\Delta,\Lambda}{,\ \to \Delta,A \supset B,\Lambda}\ \supset\text{: right}$$

  Here (by soundness and completeness of the decision procedure) the meta symbol $\to$ can be used equivalently with the language symbol $\supset$.

[8] which has been used in [14]

4.1 Languages

Now suppose we want to construct a formal language in which one can express judgments. Following the division in predicate or propositional logic, we define a language as a pair of a sets: The objects and relations, which correspond to the terms and predicates resp. of the predicate calculus

$$L = (O, R)$$

Every relation $r \in R$ has an associated arity, which is written as an exponent, if necessary. r can be interpreted as a predicate. Then the usual Boolean connectives, such as $\wedge, \vee, \rightarrow$ are *not* part of the language itself, but they are reintroduced on a meta-level later on, which aims at restricting the language to the expressions necessary for our purpose and hand on problems in theorem-proving for full predicate calculus to the inference machinery.

The judgments expressible in the language then are terms of the following structure:

$$J_L := \{r^{(n)}(o_1, \ldots, o_n) \mid o_1, \ldots, o_n \in O \wedge r^{(n)} \in R\}$$

Now judgments may be combined using the usual connectives $\wedge, \vee, \rightarrow$ with the usual semantics; the resulting expressions are the *formulae* F_L of the language L.

The relations in R have to be characterized *outside* of the language and may not be altered *within* the language. This definition will most probably be done in terms of (non-deterministic) inference rules. In this way, the properties of the relation may be hard-wired into a decision procedure, which is also the usual form of implementing type systems.

For this description, we will have to introduce some sort of *context*, which is a set of *assumptions* considered valid. These assumptions are formulae over the judgments (in particular, they can include $\rightarrow$, effectively introducing preconditions). The inference rules contain the procedures to apply them. Generally speaking, a decision procedure for a language L will have to determine if (and why) a judgment $j \in J_L$ is valid in a context , , which we write as

$$, \models j$$

Note that syntactically this language seems to be more constraint than first-order predicate calculus, since the user cannot introduce new relations (=predicates) directly and axiomatize them (by sentences of the logic). However, it is well possible to have a language $L_P = (O_P, R_P)$ where O_P contains all the predicate symbols one wants to use and R_P incorporates a sound and complete decision procedure for first-order formulae.

4.2 Reduction

Given two languages $L_1 = (O_1, R_1)$ and $L_2 = (O_2, R_2)$, whose sets of judgments are denoted by J_1 and J_2, a reduction is a triple of mappings $\Phi = (\Phi_O, \Phi_R, \Phi_\Gamma)$ with

$$\begin{array}{lcl} \Phi_O & : & O_1 \to O_2 \\ \Phi_R & : & R_1 \to R_2 \\ \Phi_\Gamma & : & \mathcal{P}(F_1) \to \mathcal{P}(F_2) \end{array}$$

such that $\forall(o_1, \ldots, o_n \in O_1, r^{(n)} \in R_1, , \ \subset F_1)$

1. $\Phi_R(r^{(n)}) = r'^{(n)} \in R_2$
2. $, \ \models r(o_1, \ldots, o_n) \iff \Phi_\Gamma(,) \models \Phi_R(r)(\Phi_O(o_1), \ldots, \Phi_O(o_n))$

Such reductions are useful for two purposes:

- A given problem can be solved with a known *more general*, i.e. more powerful technique.
- Bootstrapping: Having a simple language available, which unfortunately cannot be used easily, one can define "syntactic sugar" which then is expressed in terms of the implemented language.

We will use both sorts of reduction: A simple, compilable language will be used to give the semantics of a programming language (see Mini-Haskell), on the other hand translating judgments to Horn-clauses can help experiment easily with available theorem-provers (in our case OTTER).

Another goal is to compare our type system to several others. One example has already been alluded to in translating the co-/contravariance properties to the relation $<$ only.

If L is our system and L' some other principle, e.g. an object-oriented system with a class hierarchy, then there are two important points to be fulfilled by a reduction $\Phi : L' \to L$:

1. Φ is simple in some sense. Since L will be computationally complete, it is obvious that any other implementable system can be reduced to L, i.e. as a program, which implements L'. However, this does not allow any statements about whether our system captures the essential features of the other one.
2. If L' is decidable, then $\Phi(L') \subset L$ should be decidable (although L may not be).
3. If a judgment $j \in J_{L'}$ can be decided with algorithmic bounds $O(f)$, then $\Phi(j) \in J_L$ can be decided with the same bounds.

5 A Unifying Approach

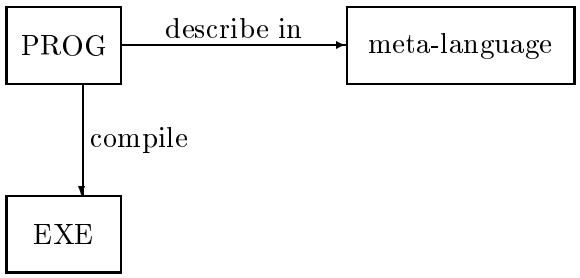
In this section of the essay we are going to describe a language L_{TE} (for *t*yped *e*xecution), whose judgments can be used to model programming language constructs as shown in section 7. These constructs not only imitate traditional type checking, but extend it to support the desirable constructs introduced in section 2.

There are essentially two ways of implementing the language L_{TE}: We can introduce proof-trees as in natural deduction with rules for the only judgment $\rightarrow$ or find a reduction to Horn-clauses and compute some examples. Resolution together with appropriate search strategies ensures breadth-first traversal of the inference tree and therefore completeness: Whenever the program is legal, then for example type checking will terminate and yield a proof containing the information, why the program is regarded correct.

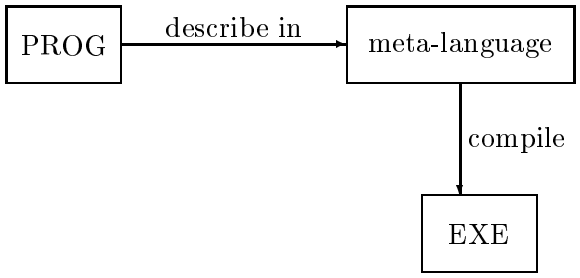
However, in this approach we do not get

- precise error messages, which are essential to make a programming language
- application-specific search strategies, which might yield a more efficient type-checking process for frequent special cases.

Concerning the relation between specification, compilation and what is expressible in the language, the unfortunate situation in most programming languages is the following:



The program is written down in a programming language, which is not capable of expressing the desired specification with the required precision. Therefore, most of the crucial details of the specification have to be moved to comments for the human reader.



If now we have a meta-language available, which is capable of expressing the needed specification, then the compiler can check more of the correct usage of a piece of software and since it can be done mechanically, the specification can be enforced.

5.1 On Well-Typing

It is customary [18], [30] to follow a three step process when introducing a type system:

1. Define a pure language and operational semantics (for example the untyped λ-calculus)
2. Define what "well-typing" should mean
3. Show (as a theorem) that a well-typed program cannot "go wrong", i.e. no extra run-time type checking is necessary.

We propose to go the opposite way:

1. Define which programs run correctly with all type information evaluated at run-time.
2. Eliminate as much of the run-time checking as possible by static inferences and show that only redundant requirements are eliminated.

In particular, we do not remove any checks of values, which are in general only available at run-time, unless the values are constants. If we regard the second step as "partly running the program", then the interpreter built for the first goal is easily transformed to a compiler for the language – provided that we can influence the process of proving correctness to separate static and dynamic requirements. This cannot be accomplished when using OTTER

or similar systems, and another reason for building a specialized inference system.

Some further thoughts on correctness can be found in section 8.3.1, which bases the correctness of a whole program inductively on:

1. Correct usage of elementary operations on built-in types and type-constructors
2. Interface-correctness, which consists of the requirements
   (a) A function is called according to its interface
   (b) Its body implements the interface specification

The question, how the objects a specification talks about can be described in another implementable language and what impacts this has concerning semantical restrictions is discussed in 8.6.

We see the justification for this way of attack in the problems related with incorporation of dynamic predicates (such as primality[9] of n for $\mathbf{Z}/n\mathbf{Z}$) into existing static type systems and in the relative simplicity of compiler-construction *and* verification (compiler-correctness theorem from functional programming languages).

5.2 Hierarchical Objects

We have seen that it is useful to have several layers of objects, such as values, types and type-classes. In this section we will attempt at a further abstraction, which will prove extremely useful for later applications to overload resolution and execution.

So far, we have aimed at stating relations between objects for purposes of abstraction, for example that a type is in a class. We would write down the type as an object of level i =2 and the class of level $j = 3$. Strictly speaking, it is not necessary that $j = i+1$, i.e. the levels follow in consecutive order, as long as we don't leave out a level. Suppose we classify the different kinds of objects we want to talk about into sets O_i, where $i \in I$ and I is an arbitrary index-set (the special case is $I := \{1, 2, 3\}$). The question is: Is there a reordering of I, such that relations are only stated between $O_i \to O_j$ where $i \leq j$, i.e. using abstraction ?

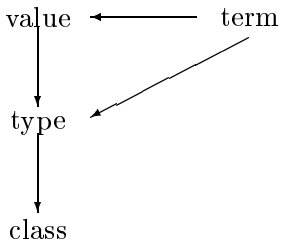
This question, of course, is answered with yes, iff the directed graph

$$N := \{\, O_i \,\}_{i \in I}, \quad E := \{\, (k, l) \mid \text{a statement } \longrightarrow_k^l \} \text{ is given } \}$$

[9] In an actual program, one would *assert* by a declaration that e.g. all numbers from a system-list are prime and hand on this attribute through interfaces.

is acyclic ([6, lemma 23.10]). Any topological sorting gives the desired renaming of I.

The following graph is the goal of this essay:



This picture has the following interpretation

- The usual hierarchy value-type-class is present.
- A term may reduce to a value via computation.
- A term may be asserted to yield a return type when executed (aiming towards compilation).

Note that we cannot go "up" the dag with any edge. This requirement of avoiding cycles can be motivated from two approaches:

- If the simple objects are arranged such that $\rightarrow$ represents some grouping into sets, what sense does it make to have objects that group "themselves" ?
- Arranging sets into a hierarchy (see [7, p. 272]) avoids the occurrence of set theoretic anomalies, and we certainly do not want them in a programming language.

A *path* in a directed graph is defined inductively:

1. () is the empty path, which contains no vertex.
2. For any $v \in N$, (v) is a path.
3. If $(v_1, \ldots, v_n)$ is a path and $(v_n, v) \in E$, then $(v_1, \ldots, v_n, v)$ is a path.

If $p = (v_1, \ldots, v_k, \ldots, v_l, \ldots, v_n)$ is a path, then $q = (v_k, \ldots, v_l)$ is a *sub-path* of p, which is written as

$$p = (p_1|q|p_2)$$

Note that possibly $p_1 = ()$ or $p_2 = ()$.

If the graph is acyclic with finite sets of vertices and edges, there are only finitely many paths, in particular every path has a finite length and the number of edges is finite.

5.3 Typed Execution

Now we arrive at the most important part of this essay: The language L_{TE} which incorporates one possible notion of generic computation.

It seems easiest to give the objects, relations and inference-rules of the language (according to section 4) and then to show what can be done with them in examples and little lemmas.

In section 5.2 we have abstracted from our hierarchy of objects to obtain a directed acyclic graph. We now make the paths of this graph the objects of the language. These have the natural interpretation, which will also be explicit in the programming language, of value, types and classes, but also values *which have* types, types which are seen *as member of a type class*, etc.

Note that these elements of the graph are *parameters* of a language, but they are not part of, i.e. expressible within, the language. If they are to be provided within a program, a meta-language must be invented, but since they carry semantic information, it is expected that they are predefined and unchangeable for a particular programming language.

Then let

$$G = (N, E)$$

be the directed acyclic graph describing the different sets of *simple objects* (types, values, etc.) as the elements of N and E the directions of abstraction/embedding/description/....

$$O_{\mathrm{TE}} := \{(x_1, \ldots, x_n) \in v_1 \times \cdots \times v_n | (v_1, \ldots, v_n) \text{ is a path in } G\}$$

If $o = (x_1, \ldots, x_n) \in v_1 \times \cdots \times v_n$ is an object and the corresponding path is $p = (v_1, \ldots, v_n)$, then we write, abusing notation:

$$(x_1, \ldots, x_n) \in (v_1, \ldots, v_n)$$

or

$$o \in p$$

On the programming language side, levels are separated by colons, and for example a value x with a type t and class c is written as

$$:_v x :_t t :_c c$$

Most of the time, since the starting point and direction is clear, we will write

$$x : t : c$$

as a readable, but context-sensitive variant. When one of the middle components is meant to be left unspecified, we write for example

$$x :: c$$

These compound language objects will come handy for three reasons:

- Although most rules will need to know only about one of the levels, conversion rules for example are an important exception: If x is a natural, then strictly speaking x is not an integer, but `nat_to_int(`x`)` is, such that the conversion rule would be stated as

  $$(x, \text{nat}) \rightarrow (\text{nat\_to\_int}(x), \text{int})$$

  Otherwise, we are restricted to the notion of subsorts, which abstracts from representation issues.

- Function calls can be expressed within the decision language, yielding a general notion of overload resolution (see sec. 7.2).

- The ambiguous situations can be solved easily. Suppose we declare the following variable:

  $$x \in \text{nat}$$

  and later on want to put x into a sorted sequence. Since there are several orderings for `nat`, we may want to make it explicit and write:

  $$\text{insert}(x :: \text{OrderedSet}(\geq), s)$$

  This expression indicates that x is the value-level, the type level is not changed and the class level is overridden by $\mathit{OrderedSet}(\geq)$, resulting in the fully annotated parse tree

  $$\text{insert}((x, \text{nat}, \text{OrderedSet}(\geq)), s)$$

if s is a sequence.

It is therefore possible to write plenty of facts into the knowledge database and still have a chance to make programs unambiguous.

In another situation, we might want to force a conversion:

$$f(x : \text{int})$$

These additional annotations must of course be checked for validity. Furthermore, to preserve the prescription of the user, we must introduce the meta-rule that prescribed values must not be inferred i.e. overridden.

The relations of the language are simply built along the edges of the given graph. It is worth noting however, that the edges of the graph are *not* judgments of the language by themselves.

If p, q is are paths of G then the judgments connected with these are

$$J_{p,q} := \{o \longrightarrow_p^q o' \mid o \in p, o' \in q\}$$

The relations of the language are therefore:

$$R_{\text{TE}} := \bigcup_{p,q \text{ paths of } G} J_{p,q}$$

There are only finitely many paths of G, therefore the union is finite. Of course the $J_{p,q}$ may be infinite, since they include user-defined objects, e.g. elements of term-algebras.

Although there may be many relations $\longrightarrow_p^q$, we usually write $\rightarrow$, because the paths are clear from the compound objects on both sides.

For the inference rules we will make use of

- *Context* , as seen in sec. 4.1 is a set of assumptions, which here are conditional rules

  $$, = \{a \rightarrow b \textbf{ where } \{d_i \rightarrow d_i'\}_{i=1}^n\}$$

  where plays the role of the logical connectives allowed in the general setting.

There are only four inference rules, which are quite general:

$$\text{[axiom]}\ \frac{\begin{array}{l},\ = \Delta \cup \{a' \to b' \textbf{ where } \{c_i \to d_i\}_{i=1}^n\} \\ \exists \tau:\ a = a'\tau \wedge b'\tau = b \wedge \\ \qquad \forall 1 \le i \le n:,\ \models c_i\tau \to d_i\tau\end{array}}{(u|a) \to (u|b)} \quad \forall (u|a), (u|b) \in O_{\mathrm{TE}}$$

$$\text{[trans]}\ \frac{A \to B \quad B \to C}{A \to C} \quad \forall A, B, C \in O_{\mathrm{TE}}$$

$$\text{[forget]}\ \frac{}{,\ \models (p|q) \to (p)}$$

$$\text{[init]}\ \frac{,\ \models X\sigma \to Y\theta}{,\ \models X \overset{\sigma,\theta}{\to} Y}$$

Remarks:

- The main ideas from section 3 are present:
  - $\to$ is a preorder
  - Substitutions are possible
  - They are computed by unification, allowing for contravariant positions (see page 20)
- To be able to modify the start and endpoint of an inference by suitable substitutions, we introduce the special symbol $\overset{\sigma,\tau}{\to}$ with two substitutions. Note that [init] can be applied only once at the root of the proof tree, since all the other judgments involve only $\to$.

  We refer to the given σ, τ as the *initial substitutions* of the judgment. These can be used to instantiate variables on the *left* hand side of a judgment, which we identified in section 3.8 as a major obstacle in introducing contravariant behaviour.
- The substitution τ in [axiom] generates an "appropriate" instance of a rule in , . Since it is also applied to the left hand sides of the preconditions, formally we need only $\to$ instead of the more general $\overset{\sigma_i,\theta_i}{\to}$ when checking those without losing expressiveness. Also, the τ_i may contain a substitution for variables $x \in \mathrm{Vars(b')} - \mathrm{Vars(a')}$; this specialization may then be used in some other part of the proof tree.
- Checking the conditions , $\models c_i \to d_i$ of an axiom merely requires building another proof-tree recursively, i.e. no structurally new judgments arise.

- In constructing a proof-tree, there is a lot of guesswork involved, since we need to pin down
  - the intermediate B for [trans]
  - the local specializations τ
  - the initial substitutions σ, θ

  We devote section 5.4 to finding a deterministic way of computing these elements of a proof-tree. Essentially, we have to give an incremental way of finding the substitutions from the requirements encountered during construction. Note the similarity to the problem of finding a most general unifier in the UNIFICATION THEOREM [25, p.33].

- [forget] contains reflexivity of $\rightarrow$ for $q = ()$.

- The [forget] rule is used to shorten an objects tail, which corresponds simply to a loss of information, and doesn't really contribute to the proof. Unfortunately it can clutter up a tree considerably. Suppose we have proof-trees both $\boxed{A \rightarrow B}$ and $\boxed{B' \rightarrow C}$ and want to prove $A \rightarrow C$. Now B is a bit longer than B', so that we need to use [forget].

$$[\text{trans}]\dfrac{\boxed{A \rightarrow B} \quad [\text{forget}]\dfrac{\overline{B = (b|b') \rightarrow (b) = B'}}{A \rightarrow B'} \quad \boxed{B' \rightarrow C}}{A \rightarrow C}$$

  In section 5.4 we will therefore consider a simpler but equivalent system with modified [reflex], [trans] and [axiom] rules.

- In later sections we will unroll the "hidden" influences of the τ and σ on *later* [axiom] rules in the proof tree. We believe that the presentation with "guesswork" is clearer and the principle ideas become more obvious. The tedious details of which *local* unification steps influence which previously computed substitutions would rather obstruct the simple underlying scheme.

5.4 Decision Language : Inference-trees

We have abstractly defined by now a system for expressing type-correct execution. However, we are missing an efficient way of constructing a proof tree without too much "guesswork". This will be done in three steps:

1. Eliminate the need for [forget] by incorporating it to the other rules.

2. Linearize proof-trees by considering only the leaves, i.e. the trees without [trans] and [init]

3. Find the substitutions incrementally by local unification.

5.4.1 Eliminating the [forget] rule

For facilitating proofs, we replace our system by the following, equivalent one. The [forget] rule is incorporated into the other rules and itself degenerates to the [reflex] rule, which used to be a special case of [forget]. We have given the other system first, because we believe that reflects better the principal ideas.

The new axiom rule is a concatenation of [forget],[axiom] and [forget], [trans] now incorporates two [forget] rules.

$$[\text{axiom}]\ \frac{\begin{array}{l}, \ = \Delta \cup \{a' \to b' \textbf{ where } \{c_i \to d_i\}_{i=1}^n\} \\ \exists \tau: \ \ a = a'\tau \wedge b'\tau = (b|b_2'') \\ \qquad \forall 1 \le i \le n: , \ \models c_i\tau \to d_i\tau\end{array}}{(u|a|u_2) \to (u|b)} \ \forall (u|a|u_2), (u|b) \in O_{\text{TE}}$$

$$[\text{trans}]\ \frac{a \to (b|b_1) \quad b \to (c|c_1)}{(a|a_1) \to (c)} \quad \forall A, B, C \in O_{\text{TE}}$$

$$[\text{reflex}]\ \frac{}{, \ \models A \to A}$$

$$[\text{init}]\ \frac{, \ \models X\sigma \to Y\theta}{, \ \models X \overset{\sigma,\theta}{\to} Y}$$

Remarks:

- What we have done is to augment any application of [axiom],[trans] with a possibly trivial application of [forget].
- We need [reflex] now, because otherwise an object could not be related to itself (in the empty context, i.e. without writing rules).
- The [forget] rule can be retrieved by applying [reflex] and [trans]:

$$\frac{\boxed{\text{T}} \quad [\text{reflex}]}{[trans]}$$

  where $\boxed{\text{T}}$ is some proof tree.

- We can w.l.o.g. assume that [reflex] appears in no proof tree except of the trivial one for

$$\frac{[\text{reflex}] \quad [\text{reflex}]}{[\text{trans}]}$$

  i.e. finding the relation $(a|a') \to (a)$. In all other positions, [reflex] does not contribute to the proof.

5.4.2 Linearizing Proofs

The essence of the proof-trees are the applications of [axiom], where the rules from , are really exploited. In this section we want to consider the sequence of leaves, read from left to right, as the means of describing derivations.

In the previous section, we have given a system, in which proof-trees have a very regular structure. Since [reflex] can be assumed to appear only in trivial cases, the leaves are [axiom] rules and the inner nodes are build by [trans]. In this structure, basically in every derivation step, there is a hidden [forget]; we introduce a special notation

$$a \rhd b$$

to be equivalent to

$$a = (a_1|a_2) \to (a_1) = b$$

For the relation $\rhd$ we have the simple property: For all objects a, b, c

$$a \rhd b \rhd c \;\Rightarrow\; a \rhd c$$

The second essential step is the application of an [axiom] rule, which we write as

$$A \Rightarrow_\Gamma B$$

About global substitutions, there is a simple observation to be stated:

Lemma 1 *In the derivation system for $\to$, we have*

$$\forall \sigma, \theta : \; A \overset{\sigma,\theta}{\to} B \iff A\sigma \to B\theta$$

This follows directly from the fact, that [init] is only applicable once per proof-tree, i.e. in the root.

The following lemma seems very useful for an implementation. It says that instead of reading the whole proof-tree, it is enough to read the leaves from left to right. Therefore, the data structures for holding proof steps can be considerably simpler.

Lemma 2 *Let $X \overset{\sigma,\theta}{\to} Y$ be a judgment and , a context. Then the following two statements are equivalent:*

1. *There exists a proof-tree without the use of conditional rules for*

$$, \models X \overset{\sigma,\theta}{\to} Y .$$

2. *There are objects $\{X_i, X_i', X_i''\}_{i=1}^n$, such that $X\sigma \rhd X_1$, $X_n = Y\theta$*

$$\forall 1 \leq i \leq n-1 : X_i \rhd X_i' \Rightarrow_\Gamma X_i'' \rhd X_{i+1}.$$

where $\Rightarrow_\Gamma$ never uses a conditional rule.

Proof: We first show that the claimed sequence can be constructed from a proof tree.

By lemma 1, it is equivalent to show the claim for the relation $X' \to Y'$ where $X' = X\sigma$,$Y' = Y\theta$.

For the trivial tree proving $X' = (a|a') \to (a) = Y'$ we simply set $n = 1$ $X_1 = a$. Then $X' \rhd X_1 \rhd Y'$.

W.l.o.g. we now assume that [reflex] is not used in the tree, i.e. there are only [axiom] and [trans] rules, where exactly the [axiom] are the leaf nodes and [trans] are the inner nodes.

Structural induction gives the general result.
Base: Let X', Y' be objects with $X' \to Y'$. If the proof tree consists of a leaf, then it looks as follows.

$$[\text{axiom}]\frac{, = \Delta \cup \{a' \to b'\} \quad a'\tau = a \quad b'\tau = (b|b_1')}{(u|a|a_2) \to (u|b)} \quad \begin{array}{l}\text{for some}\\ \text{substitution } \tau\end{array}$$

Therefore $X' = (u|a|a_2)$, $Y' = (u|b)$ and with $n = 1$

$$X' = (u|a|a_2) \rhd (u|a) \Rightarrow_\Gamma (u|b) \rhd (u|b) = Y'$$

by definition of $\Rightarrow_\Gamma$.
Step: All we need to do is cut out of the middle of the two chains some redundant $\rhd$ relation.

The root of the tree now consists of an application of [trans]:

$$[\text{trans}]\frac{a \to (b|b_1) \quad b \to (c|c_1)}{(a|a_1) \to (c)}$$

Therefore $X' = (a|a_1)$, $Y' = c$. By the induction hypothesis we can find $\{A_i, A_i', A_i''\}_{i=1}^l$ and $\{B_i, B_i', B_i''\}_{i=1}^m$ with

$$(a) \rhd A_1 \rhd A_1' \cdots \Rightarrow_\Gamma A_{l-1}'' \rhd A_l = (b|b_1)$$

and

$$(b) \rhd B_1 \rhd B_1' \cdots \Rightarrow_\Gamma B_{m-1}'' \rhd B_m = (c|c_1)\,.$$

Because

$$A_{l-1}'' \rhd A_l = (b|b_1) \rhd (b) \rhd B_1 \rhd B_1'$$

and therefore by the property of $\rhd$

$$A_{l-1}'' \rhd A_l \rhd B_1'$$

Therefore, we can choose the X_i as follows:

$$X_i := \begin{cases} A_i & i < l \\ A_l & i = l \\ B_{i-l+1} & l+1 \leq i \leq l+m \end{cases}$$

Conversely, given a sequence as above, it is easy to build a proof-tree, since the operations $\Rightarrow_\Gamma, \rhd$ allowed in sequences can be emulated by the [axiom], [trans] rules only, if we handle the special case $A \rhd B$ separately as above. In this direction, we do not need to pay attention to how many $\rhd$ we use. $\Box$

Obviously, this lemma changes the structure of our proofs. If no conditional rules are allowed and if we incorporate the $\rhd$ operation into a $\bigcirc$, then a proof has this form:

$$\bigcirc \Rightarrow_\Gamma \bigcirc \Rightarrow_\Gamma \bigcirc \cdots \bigcirc \Rightarrow_\Gamma \bigcirc \Rightarrow_\Gamma \bigcirc$$

Towards an implementation, it is clear that if there is a sequence proving , $\models A \overset{\sigma,\theta}{\to} B$, then it can be build from the beginning to the end (induction on the length of the sequence) the next subsections gives the details.

5.4.3 Incremental Computation of Substitutions

In the preceeding discussion we have always assumed that the applicable inference rules and the substitutions occurring in the [init] and [axiom] rules are given non-deterministically when building a proof-tree. We will dispose of the latter requirement in this subsection, which unveils the inner dependencies between the global and local substitutions. However, we *do* assume that we know the sequence of rules to be applied. The goal is to compute the most general substitutions using a unification step locally. In fact, given a sequence $(r_1, \ldots, r_n) \subseteq$, (without conditional rules) we can retrieve the

essential requirements on the corresponding τ_i as a system of equations over the X_i'' and X_{i+1} from lemma 2.

The following definition "finds" these equations (if they exists) and respects the $\triangleright$ relation. Now we can find the equations induced by a given sequence of rules:

Definition 1 *Let A, B be sets of equations over terms. Then*

$$A \dot{\cup} B := \begin{cases} \textbf{fail} & A = \textbf{fail} \vee B = \textbf{fail} \\ A \cup B & \textit{otherwise} \end{cases}$$

Let $R = (a_i \to b_i)_{i=1}^n \subseteq$, be a sequence of unconditional rules and X, Y compound objects and $\{v_{i_j}\}$, $\{u_h\}$ nodes in the graph of classes.

Then the function $\mathbf{equ}(X, Y, R)$ *is defined inductively on the length of R:*

$\boxed{n = 0}$ *If*

$$\begin{array}{l} X = (x_1,\ldots,x_m,\ldots,x_n) \in (v_{i_1},\ldots,v_{i_m},\ldots,v_{i_n}) \\ Y = (y_1,\ldots,y_m) \qquad\quad \in (v_{i_1},\ldots,v_{i_m}) \end{array}$$

then

$$\mathbf{equ}(X, Y, R) := \{x_j = y_j\}_{j=1}^m$$

otherwise

$$\mathbf{equ}(X, Y, R) := \textbf{fail}$$

$\boxed{n > 1}$ *If*

$$\begin{array}{l} X = (x_1,\ldots, x_l, \ldots, \quad x_m, \quad \ldots,x_n) \in (v_{i_1},\ldots, v_{i_l}, \ldots, v_{i_m},\ldots,v_{i_n}) \\ a_1 = \qquad (y_1,\ldots,y_{l-m+1}) \qquad\quad \in \qquad (v_{i_l},\ldots,v_{i_m}) \end{array}$$

and

$$b_1 = (z_1,\ldots,z_k) \in (u_1,\ldots,u_k)$$

then let

$$E_1 := \{x_{s+l-1} = y_s\}_{s=1}^{l-m+1}$$

and

$$E_2 := \mathbf{equ}(\,(x_1,\ldots,x_l,z_1,\ldots,z_k)\,, Y, (a_i \to b_i)_{i=2}^n)$$

and define

$$\mathbf{equ}(X, Y, R) := E_1 \dot{\cup} E_2$$

If the above conditions are not met, define

$$\mathbf{equ}(X, Y, R) := \mathbf{fail}$$

Remarks:

- An equation system over simple terms can then be solved, if possible, by the unification algorithm [25] and this algorithm finds the most general solution possible, iff it exists.
- From the definition of **equ** we see that only those equations are included into the system, which must in any case be solved, i.e. without even looking at possible conditions of the rules involved.

The substitution resulting from the system (if it exists) is very useful, since it contains τ_i and the initial σ, θ we have been looking for.

Lemma 3 *The following statements are equivalent:*

1. *There are objects* $\{X_i, X_i', X_i''\}_{i=1}^n$, *such that* $X\sigma \rhd X_1$, $X_n = Y\theta$

   $$\forall 1 \leq i \leq n-1 : X_i \rhd X_i' \Rightarrow_\Gamma X_i'' \rhd X_{i+1}$$

   where $\Rightarrow_\Gamma$ *never uses a conditional rule. (see lemma 2)*

2. *There is a sequence* $R = (a_i \to b_i)_{i=1}^n \subseteq$, *of unconditional rules with* $\forall i \neq j : (\mathrm{Vars(a_i)} \cup \mathrm{Vars(b_i)}) \cap (\mathrm{Vars(a_j)} \cup \mathrm{Vars(b_j)}) = \emptyset$ *and* X, Y *compound objects for which* $\mathbf{equ}(X, Y, R)$ *has a (most general) solution.*

Furthermore, the two forms of a proof can be constructively obtained from one another.

Proof: (1) $\Rightarrow$ (2): Since each $\Rightarrow_\Gamma$ stands for one application of axiom, the given proof induces a sequence of (unconditional) rules $(a_i \to b_i)_{i=1}^n \subseteq$, together with suitable substitutions τ_i. W.l.o.g. we can assume that $\mathrm{dom}(\tau_i) \subseteq (\mathrm{Vars(a_i)} \cup \mathrm{Vars(b_i)}$ (since τ_i is applied only locally) and after a renaming $\forall i \neq j : (\mathrm{Vars(a_i)} \cup \mathrm{Vars(b_i)}) \cap (\mathrm{Vars(a_j)} \cup \mathrm{Vars(b_j)}) = \emptyset$. Because then

variables of different rules are disjoint and the τ_i, σ and θ are substitutions (i.e. the variables do not occur in the respective terms)

$$\gamma := \sigma \cup \theta \cup \bigcup_{i=1}^{n} \tau_i$$

is a substitution. Also γ solves $S := \mathbf{equ}(X, Y, (a_i \to b_i)_{i=1}^n)$: The system S of equations over simple objects (terms) comprises by definition of **equ** only parts of the X_i, X_i' and X_i'' which are equal up to $\rhd$ and the application of the respective $\tau_i \subseteq \gamma$.

(2) $\Rightarrow$ (1): Now let $R = (a_i \to b_i)_{i=1}^n \subseteq$, given and γ the solution of $S := \mathbf{equ}(X, Y, (a_i \to b_i)_{i=1}^n)$.

We set the desired substitutions

$$\begin{aligned} \sigma &:= \gamma|_{\mathrm{Vars(X)}} \\ \theta &:= \gamma|_{\mathrm{Vars(Y)}} \\ \tau_i &:= \gamma|_{\mathrm{Vars(a_i)} \cup \mathrm{Vars(b_i)}} \end{aligned}$$

From the definition of **equ** we can find the X_i', X_i'' and the desired equalities hold, since S comprises the necessary equations by definition of **equ**.

Between the applications of [axiom] in $\Rightarrow_\Gamma$, we find the following sequence of $\rhd$ operations[10]

$$X_{i-1}'' \rhd X_i \rhd X_i'$$

which can be shortened to

$$X_{i-1}'' \rhd X_i'$$

Therefore we can simply choose $X_i := X_i'$. □

5.4.4 Non-deterministic Pathfinding

Although after the last subsection it is clear how to verify that a sequence of rules is a proof (or determine why it fails if it is not), there is still a source of non-determinism left in the approach: Of course we could simply say that the set of all sequences of rules is enumerable and then check them one by one — but this is very costly, and most sequences will fail locally, i.e. the

[10] Recall that this has been chosen to be redundant, since we had to emulate all possible [forget] steps in the proof trees (see page 32)

right hand side of a rule doesn't even "look like" the left hand side of the following rule. However, given a binary predicate `quickReject` on terms, which returns `true` if the terms would never unify, and determines this very quickly, we can build a tree of paths, where every node represents one of the X from the definition of **equ** and the children are possible continuations with different rules from the data base, whose left hand side and term of the node agree w.r.t. `quickReject`, applied.

5.4.5 Generalization to Conditional Rules

As a consequence of the preceding subsections, we conclude that given terms X and Y, we can determine all proofs with length smaller than a given N, as long as there are no conditional rules involved. However, remarks 5.4.3 are already geared towards a top-down approach to conditional rules, i.e. use the three steps

1. Find a path
2. Verify path
3. Iteratively prove all conditions occurring on the path with the substitutions found so far applied.
4. Apply the substitutions found in the recursion step to the path.

There are two important remarks to be made concerning completeness of the search process:

Remarks:

- It is easy to find reasonable examples of data bases, where (3) involves an infinity recursion, for example if accidentally no *applicable* rules without preconditions are given for a occurring judgment. Therefore, the checking of preconditions is not a simple recursive procedure, but must be incorporated into the general breadth-first search for trees.
- There may be more than one proof for a judgment, i.e. the solutions possibly found in (3) involve copying of the parent proof-sequence before (4).

It should be possible to choose the order of operations, such that "short" proofs come out first. For example, one could give priority to proofs with minimal overall length, i.e. the length of the path plus the sum over all lengths of subproofs induced by conditions.

5.4.6 Selecting the Best Proof

Since we can obtain all proofs, possibly only one by one if there are infinitely many, for a judgment, we assign weights to each of them and finally select the lightest. Such decisions would in general be used to minimize run-time penalties of needed predicate calls (for verifying a dynamic condition not statically determined by an invariant). Furthermore, there could be hierarchies on functions in the sense that preferences could be stated, such that if calls to several functions are possible in the same position of an expression tree, the one with the highest precedence is actually chosen ($\approx$ C^{++} template specialization).

Since most of the programming language constructs and also the objects one would be able to talk about, are also available in existing programming languages, we do expect that most judgments have only finitely many proofs. In this situation it is not difficult to accumulate them and afterwards find the best proof w.r.t. the given priorities. If the sequence of proofs doesn't stop by itself, there are several possible strategies:

1. Set an artificial breakpoint and don't accept proofs with total length longer than a given N. This works only if the proofs are guaranteed to come out monotonically non-decreasing. The actual threshold could be fine-tune by `pragma`s for every program part, possibly single statements.

   A warning is issued when the stop is forced. If this is a rare case, the programmer won't have troubles in examining the compiler's decision, which will in any case correct, but possibly not optimal.

2. Issue an *error* if there are proofs not examined yet and make the user specify with `AS` clauses (see section 7.12) her intentions more clearly until no long proofs occur.

   In fact, another `pragma` makes this choice a strict variant to the first strategy.

However, there is an important special case where the initial judgment , $\models A \overset{\sigma,\theta}{\rightarrow} B$ has the following properties:

- A does not contain variables.
- Only one rule in , applies to A and the resulting A' has this property, too.

This automatically yields a deterministic proof in the following cases, in which the set of rules ,

- is an encoding of deterministic inference system e.g. the λ-calculus
- describe a simple type system, e.g. OSA
- describe a program, where no overloading and coercion is used in (it may use instantiation, though)

5.5 Language Variations/Extensions

5.5.1 Run-time Checks

So far we have considered an interpreter-approach for typed-execution of programs. To get a compiler out of it, we need to draw a line of separation between compile-time and run-time, or static and dynamic decisions. Because of the generality of L_{TE}, this choice is (almost) arbitrary.

A similar situation arises in the technique of partial specialization, where we simply declare some of the input parameters static and some dynamic and let the interpreter run, until it cannot make any further reductions, because dynamic values are needed.

It seems appropriate to use the following guide-lines (recall that we have divided up the simple objects into classes):

- Most classes contain objects, which are not themselves created at run-time. The aim is to introduce a single class, i.e. a node in the graph of simple objects, named *values*, whose objects are the arguments and results of computations at run-time. If they happen to be known statically, however, they can well be evaluated by the inference process directly and *without implementation overhead*.
- There is a class *predicates*, which contains symbolic expressions for predicates on *values*. These predicates exist twice: As terms just as in L_{TE} and as executable boolean functions, which can be inserted by the compiler, where necessary. Example:

  ```
  DEF odd(x:int) := ((x%2)=1)
  ```

- For typed execution, the context must reduce a function call directly to the expression, which defines the function, e.g.

  ```
  +(x:int,y:int) -> int_plus(x,y)
  ```

  to resolve overloading for `+`. Now we rather reduce to a special term:

```
+(x:int,y:int) -> CALL(int_plus(x,y))
```

`CALL` itself will not be further reduced, but the compiler will know that a function call is to be emitted. This is only a minor change in the translation from our programming language to the decision language L_{TE}.

Furthermore, it would be helpful to give a second name to a function which uniquely identifies it[11]. The overload resolution consists of applying rules to an expression until only unique names appear.

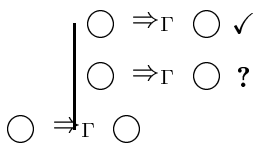
The compilation of an expression, which could be evaluated within L_{TE}, proceeds in three steps.

- Translate the expression to L_{TE}
- Construct a inference sequence for the expression.
- Eliminate as many proof-obligations as possible, i.e. prove most of the judgments of a conditional rules.
- Emit an error message, if there are proof-obligations which cannot be expressed by dynamic predicates.
- Generate code from the `CALL` terms and insert the executable parts of the objects from the *predicate* class according to the inference-sequence.

If carried out in this generality, the procedure not only yields automatic exceptions when a specification such as `odd(x:int)` is violated. It also allows us to defer any decisions concerning the program-flow to the evaluation of run-time predicates:

- overload resolution (for correctness)
- choosing the more specialized function (for efficiency)

The main idea can be visualized as follows: Suppose somewhere within the inference sequence, we use a conditional rule:



[11] If this name is compiler-generated according to a special function, this process is called "name mangling".

One of the proof-obligations can be checked statically and is marked, the other proof obligation must be expressed by dynamic predicates.

5.5.2 Binary Predicates

Making restrictions on single values, types, etc. is what is possible in most programming languages to some degree. The specification of relations between several objects can be embedded in L_{TE} most naturally within the *predicates* class, too. Again, the same threads of reasoning about static/dynamic decisions can be carried out, with the same result: The border line is arbitrary and the solution is general.

5.6 The Problem of Error-Messages

Doing inference does not automatically yield an intelligible description of why and where a proof failed or also, in case the user wishes to understand better his program's behaviour, to explain the decision taken.

This requires dependencies between messages from the system, which can be expressed very well in a language with hyper-links, such as HTML. In our view, a good system should at least provide the following:

- Failure/Success in a message part (i.e. a hyper-link) is visible without following it.
- There are different symbols attached to links, according to the meaning, e.g. message A explains B, B is one possible inference from A.
- Links are used meaningfully to structure the level of detail that is given in the error-message.
- The generated structure is not purely hierarchical, but there are cross-references and short-cuts to messages at as deeper nesting level. Heuristics about frequent errors can be used to guide the user.

6 The Otter Proof System

OTTER[17] is a Horn-clause theorem prover with additional features which lead towards logic programming, such as demodulation (rewriting) of terms, built-in functions and predicates for boolean and machine-precision integer values and bit-fields.

Why we think that using this tool for experiments could be helpful, can be justified with four observations:

- Unlike in term-rewriting,
  - non-determinism of the set of rules is expected and supported (breadth first search)
  - matching (unification) takes place only at the topmost level and not in the middle of terms (which is the main reason why [5] must assume that type-constructors are covariant)
- The τ in our [axiom] rule can be determined by repeated unification (yielding the most general unifier, which can be further instantiated afterwards). OTTER provides this tool and also applies τ to the whole rule as defined by [axiom].
- Horn clauses with exactly one positive literal can be regarded as rules with one conclusion and several preconditions, just as in our rules.
- OTTER is extremely fast. All examples except those containing actual computation, were checked in less than 50ms (!) on a Pentium/266 (64MB RAM). and even larger data-bases with mostly unusable rules won't do too much harm, because of the built-in term-indexing scheme.

The encoding of $\rightarrow$ is done by having one single predicate in the input files, called `rel`. The objects, which are arguments to the predicate, are represented by terms of the structure

```
o$(x,t,c)
```

where x represents a value, t a type and c a class. Since OTTER has no idea about our [forget] rule, we must emulate it and use distinguished constants `no_c` for "no class given" and `no_v` for "no value given" at the c and x positions resp.

Terms, which represent computations to be done, are written as

```
tm$(f, x1, ..., xm)
```

where m is the arity of the operator f. In principle, we could have enlarged the `o$()` expression and introduced `no_t`, but we think that this, although it is not a direct translation of $\rightarrow$, is more readable in the example codes.

The parser of the programming language simply needs to count arguments to generate the appropriate terms !

Therefore, there is conceptually a two step translation

1. From the program code
2. to the $\rightarrow$ calculus
3. to the Ottter input

One drawback of this approach is that at the very end, we cannot tell OTTER which literal is to be resolved first in a clause, which might give many "undesired" and "useless" inference steps for *our particular application*.

To keep the examples short, we give the structure of the otter files we used here:

```
set(neg_hyper_res).
set(pretty_print).
set(hyper_res).
set(prolog_style_variables).
clear(back_sub).
clear(for_sub).
assign(max_proofs, 10).
assign(max_seconds, 20).

list(usable).

 % put the general inference rules here

end_of_list.

list(sos).

 % ask the current question here

end_of_list.
```

We found that the choice of inference techniques (binary-,hyper-, negative-hyper-resolution) had an impact on the run-time for the proof, but not on completeness (basically because we assume the functionality of binary resolution only). However, it turned out that binary resolution tended to generate many more useless and redundant inferences and the other two techniques sometimes stopped with an empty set of support for wrong programs.

A typical inference rule we would write looks like this:

```
% preconditions and evaluation of parameters
-rel(_X1, o$( .... )) |
  ...
-rel(_Xn, o$( .... )) |
% object inference
rel(o$( ... ), o$( ... ))
```

where the preconditions correspond directly to the conditional judgments of the $\rightarrow$-system.

If there is exactly one positive and one negative literal in the clause, it can be viewed providing a possible replacement of a precondition or conclusion of another rule. However, in such a resolution, the other clause will not be shortened, i.e. a proof is not found. This can only be achieved using a single positive literal, and this construct can be read as a (programming language) variable/constant/... declaration.

7 Translation of Programming Language Constructs

In this section we show how to express several programming language constructs, such that they can be easily combined. This can be achieved by using several classes of simple objects. The graph pertaining to the language is build step by step. We start with the general class *term*, which is attached to terms by the parser. They represent computations yet to be evaluated (at run-time). A second class is *value*, which contains both integers and booleans (which are treated as built into the system). The values will be extended to contain value-constructors in section 7.8. Also, terms containing only unique function names with checked parameters are regarded as values, since they could be reduced using rewriting. The next two classes of the sequence are *type* and *type class*, which serve to identify the type of values and the (more abstract) group, which the type belongs to (e.g. Integers, seen as a monoid with + and 0).

In the following we will introduce a programming language with the features considered useful especially in the field of computer algebra. Although we are not aware of any existing language supporting all of them, we think that with the approach presented here, i.e. a reduction to L_{TE}, they can be implemented and a compiler to a simple untyped language can be built (see section 8.3.1). For the description of the single constructs we will use terminology taken from literature on programming languages.

For the sake of readability, we do not follow the recursive definitions fully (i.e. in an systematic way), for example we abbreviate

```
o$(_l, list(o$(no_v, int, no_c), no_c)
```

as

```
o$(_l, list(int), _c2)
```

knowing that `int` stands for some type in this syntactic position. This is intended just for the human reader, but could obviously avoided in a formal translation.

7.1 Notation for L_{TE}

The rules of L_{TE} have a simple structure, which we want to emphasize by using a special notation. In section 5.3 we have already introduced the objects of the language together with : as a meta-symbol for notational convenience. For example we write

$$:_{C_1}\ x_1\ :_{C_2}\ x_2 \ldots :_{C_k}\ x_k$$

for a compound object

$$(x_1, x_2, \ldots, x_k) \in C_1 \times C_2 \times \cdots \times C_k$$

where $(C_1, C_2, \ldots, C_n)$ is a path in the graph of classes.

Since from the context of the examples it will be clear what path is taken, we omit the explicit subscript of : and write

$$x : t : C$$

if x is a value of type t in class C. This simplification is further supported by the usual convention of naming variables:

- Values are at the end of the latin alphabet: $x, y, \ldots$.
- Types are denoted by lower-case s, t.
- Classes are upper-case letter at the beginning of the alphabet: $A, B, C, \ldots$.

The terms do not fit well into this scheme, and so we do *not* abbreviate them, writing:

$$:_T\ f(x_1, \ldots, x_n)$$

A rule, which can be used during an application of the [axiom] inference rule, has the structure:

$$A \to B \quad \Big| \quad \begin{array}{l} P_1 \to Q_1 \\ \quad \vdots \\ P_n \to Q_n \end{array}$$

Here $A \to B$ is the main relation between objects A and B and the $P_i \to Q_i$ are preconditions. The variables are supposed to be quantified at rule level, according to the definition of [axiom].

During the description to follow, we will give the required rules in both the above short-hand notation and the translation to the OTTER[17] system and briefly indicate how the rules can be obtained systematically as a (simple, syntactic) translation from the programming language.

7.2 Function Calls and Overload Resolution

The goal is to provide an expression of declared functions, which in most languages have a form similar to

```
function name(x1:t1, ... xn:tn) : t = term
```

where `term` is the body of the function, `name` identifies the function (probably it is overloaded) and the x_i are parameters of type t_i. This can be reinterpreted in a sentence like

> If x_i is of type t_i for all $1 \leq i \leq n$, then a term `tn$(name, x1, ..., xn)` can be replaced by `term` and evaluating the resulting expression yields a value of type `t`.

For obtaining compilation, `term` would be an mangled version of the overloaded `name`. The unique identifier does itself not appear on a left hand side of a rule.

Note further that assumptions could be recursively stated about the t_i such as in

```
f(x:list(alpha:orderedSet(p:(alpha,alpha)->bool))):t(alpha)
```

which would be readily "linearized" or unrolled with a simple recursive function as:

> If x is a list over a type α and α is an ordered set with parameter (relation) f, and f is a function from $\alpha \times \alpha$ to bool, *then* f can be evaluated to some value of type $t(\alpha)$.

If the unique (mangled) name for f is F, the syntactic translation required to obtain an L_{TE} rule, would yield:

$$:_T f(x) \to F(x) : t(\alpha) \quad \Big| \quad \begin{array}{l} \alpha \to \mathrm{orderedSet}(p) \\ p \to p' : ((\alpha, \alpha) \to \mathrm{bool}) \end{array}$$

Here $\to$ is the function space (type-) constructor and p is the predicate realizing an order.

> The function calls in the programming language are characterized by
>
> - call-by-value $\Rightarrow$ eager evaluation
> - one return value
> - implicit coercion (and subsorts) (see sec. 7.4)
>
> Section 7.10 discusses how to include reference-parameters naturally and obtain the desired constraints (no conversions apply).

An example with parameterized data types is given by a modular "/" function. The predicate `is_prime` is easily written down in OTTER, but not in the calculus so far. Of course, it is infeasible to implement a run-time predicate for primality testing; we have chosen the example nevertheless to give a motivation for section 8.5.

The programming language declaration

```
function '/'(x:mod(n:PrimeInt), y:mod(n:PrimeInt)):mod(n).
```

can be written down in L_{TE} as follows:

$$:_T /(x, y) \to \mathrm{div\_mod}(n_1, x', y') \quad \Big| \quad \begin{array}{l} x \to x' : \mathrm{mod}(n_1) \\ y \to y' : \mathrm{mod}(n_2) \\ n_1 \to n_1' : \mathrm{Int} \\ n_2 \to n_2' : \mathrm{Int} \\ \mathit{INT\_EQUAL}(n_1', n_2') \\ \mathit{PRIME\_INT}(n_1') \end{array}$$

Note that the main deduction makes the type-parameter n a parameter to the primitive procedure.

The first four preconditions serve two purposes:

1. Evaluate parameters (or obtain translation when compiling)

2. Type-check by requiring a certain result

In OTTER, the example can be written almost identically. Recall the we use the predicate `rel` to denote $\rightarrow$.

```
%%       a -> (x, mod(n1))
   -rel(_a, o$(_av, mod(_n1))) |
%%       b -> (y, mod(n2))
   -rel(_b, o$(_bv, mod(_n2))) |
%%       n1 -> o$(n1', int)
   -rel(_n1, o$(_nn1, int)) |
%%       n1 -> o$(n1', int)
   -rel(_n2, o$(_nn2, int)) |
%% Extra attributes
   -is_prime(_nn1) |
   -$NOT($NE(_nn1,_nn2))
%% consequence is evaluation of a term
 | rel(t2$(div, _a, _b), o$(div_mod(_nn1,_av, _bv), mod(o$(_nn1,int)))).
```

We had to trick OTTER by writing `$NOT($NE(_nn1,_nn2))`, because it knows about equivalence, and would unify `_x` and `_y` in `$EQ(_x,_y)` which is not the desired effect: The two terms must evaluate to the same object, but they need not be identical.

Writing `o$(div_mod(_nn1,_av, _bv), mod(o$(_nn1,int))` as the consequence of the rule states two things:

- The return value is `mod(o$(_nn1,int))`
- The function can be evaluated by *reducing* (without types) `div_mod(_nn1,_av, _bv)`

  This means that overloading has been resolved, assuming that `div_mod` is a unique name and no recursive nesting of parameters occurs in this call, so the expression has been simplified and can be treated by a simple rewriting system or for code generation (annotated syntax tree!)

Of course, we have to say how the constants `t0$` are evaluated and write[12], for example:

$:_T\ 2 \rightarrow 2 : \mathrm{Int}$

or in OTTER

[12] Overloading is possible !

```
%% evaluate constants
 rel(t0$(2), o$(2,int)).
 rel(t0$(3), o$(3,int)).
 rel(t0$(5), o$(5,int)).
```

Now assume the user has given local variables and a simple term:

```
var a : mod(2+3).
var b : mod(5).
 EVAL a/b.
```

When evaluating the term a/b, there are additional things to be computed, for example $2+3$, which in an environment of overloadable constants is by no means "trivially" 5 and of type integer !

It turns out though, that *all of these considerations* can be automatically found and handled by the L_{TE} rules given so far. All that the parser needs to do to create a judgments upon finding `EVAL` is to expand the definitions of a and b. Strictly speaking, even this is not necessary: We could instead have given the rules

$$:_T a \to \text{mangled\_a} : \mathrm{mod}(2+3)$$
$$:_T b \to \text{mangled\_b} : \mathrm{mod}(5)$$

and make this expansion automatic.[13] To help OTTER a bit, the expanded version was used:

```
%% div(a:mod(2+3), b:mod(5)) ->? val : type
  -rel(t2$(div, o$(a, mod(t2$(plus, t0$(2), t0$(3)))),
                o$(b, mod(t0$(5)))),
       o$(_ret, _t))
    | $ans(_ret, _t).
```

The proof found by OTTER is

```
---------------- PROOF ----------------
1 []
rel(o$(_x,_t),o$(_x,_t)).
2 []
is_prime(5).
```

[13] The decision depends on whether we regard variable declarations as some program-meta-structure, which must accordingly handled by the parser, or as just another language construct to be translated to L_{TE}.

```
3 []
-rel(_a,o$(_av,mod(_n1))) |
-rel(_b,o$(_bv,mod(_n2))) |
-rel(_n1,o$(_nn1,int)) |
-rel(_n2,o$(_nn2,int)) |
-is_prime(_nn1) |
-$NOT($NE(_nn1,_nn2)) |
rel(t2$(div,_a,_b),o$(div_mod(_nn1,_av,_bv),
                      mod(o$(_nn1,int)))).
4 []
rel(t0$(2),o$(2,int)).
5 []
rel(t0$(3),o$(3,int)).
6 []
rel(t0$(5),o$(5,int)).
7 []
-rel(_a,o$(_av,int)) |
-rel(_b,o$(_bv,int)) |
rel(t2$(plus,_a,_b),o$($SUM(_av,_bv),int)).
8 []
-rel(t2$(div,o$(a,mod(t2$(plus,t0$(2),t0$(3)))),
             o$(b,mod(t0$(5)))),o$(_ret,_t)) |
$ans(_ret,_t).
9 [binary,8.1,3.7]
$ans(div_mod(A,B,C),mod(o$(A,int))) |
-rel(o$(a,mod(t2$(plus,t0$(2),t0$(3)))),o$(B,mod(D))) |
-rel(o$(b,mod(t0$(5))),o$(C,mod(E))) |
-rel(D,o$(A,int)) |
-rel(E,o$(F,int)) |
-is_prime(A) |
-$NOT($NE(A,F)).
109 [binary,9.1,1.1]
$ans(div_mod(A,a,B),mod(o$(A,int))) |
-rel(o$(b,mod(t0$(5))),o$(B,mod(C))) |
-rel(t2$(plus,t0$(2),t0$(3)),o$(A,int)) |
-rel(C,o$(D,int)) |
-is_prime(A) |
-$NOT($NE(A,D)).
166 [binary,109.1,1.1]
$ans(div_mod(A,a,b),mod(o$(A,int))) |
```

```
-rel(t2$(plus,t0$(2),t0$(3)),o$(A,int)) |
-rel(t0$(5),o$(B,int)) |
-is_prime(A) |
-$NOT($NE(A,B)).
186 [binary,166.2,6.1]
$ans(div_mod(A,a,b),mod(o$(A,int))) |
-rel(t2$(plus,t0$(2),t0$(3)),o$(A,int)) |
-is_prime(A) |
-$NOT($NE(A,5)).
189 [binary,186.1,7.3]
$ans(div_mod($SUM(A,B),a,b),mod(o$($SUM(A,B),int))) |
-is_prime($SUM(A,B)) |
-$NOT($NE($SUM(A,B),5)) |
-rel(t0$(2),o$(A,int)) |
-rel(t0$(3),o$(B,int)).
194 [binary,189.3,4.1]
$ans(div_mod($SUM(2,A),a,b),mod(o$($SUM(2,A),int))) |
-is_prime($SUM(2,A)) |
-$NOT($NE($SUM(2,A),5)) |
-rel(t0$(3),o$(A,int)).

196 [binary,194.3,5.1,demod]
$ans(div_mod(5,a,b),mod(o$(5,int))) |
-is_prime(5).

197 [binary,196.1,2.1]
$ans(div_mod(5,a,b),mod(o$(5,int))).
------------ end of proof -------------
```

The last inference from 196 removes the last precondition, the primality of 5 and yields the expected annotated parse tree `div_mod(5,a,b)`, which could be passed to a code generator or reduction system.

7.3 Subsorts

When computing in term algebras, a subsort $s' < s$ contains the terms constructed using a subset of the constructors available in s. However, the essential way of representation is the same, so we can easily state this fact

as: [14]

$$:_t s' \to :_t s$$

In OTTER we have to give the entire path:

```
-rel(o$(_x, _t, _c), o$(_x, s', _c)) |
 rel(o$(_x, _t, _c), o$(_x, s, no_c)).
```

This rule says that we can simply change the *tag* of value `_x`. Note that this can be done also *if s has parameters*, and we can even (recursively) impose conditions on the parameters.

If we can have conditions, however, we can even insert *retracts* (cf. [10]). For example, for the relation between naturals and integers we get the *two* rules:

$$x : \text{nat} \to x : \text{int}$$
$$x : \text{int} \to x : \text{nat} \quad | \quad \text{HTBN(x)} \to \text{TRUE}$$

where *HTBN* is a run-time predicate (inserted automatically into the code if this rule is used, see sec. 5.5.1) and the abbreviation stands for "Happens To Be a Natural".

An important phenomenon called "structural coercion" and treated by Weber [31, sec. 4.2.5] extensively, can also be modeled in L_{TE}. In particular, using the observations from section 3.8, we can even express *contra*variant parameter positions in the following way:

Suppose type constructor[15] T is contravariant in a parameter (which is the only one for ease of presentation). In the programming language, this could be naturally expressed by saying:

```
declare variance T((-)s).
```

All we have to do is write

$$:_t T(s) \to :_t T(t) \quad | \quad t \to s$$

For the covariant case, we would obtain the rule

```
declare variance T((+)s).
```

and

$$:_t T(s) \to :_t T(t) \quad | \quad s \to t$$

One could think of making covariant behaviour a default, since this seems to be the most frequent case in practice.

[14] Note that here the paths start at the *type* level.

[15] In fact, value or type-class constructors could be used as well, since L_{TE} doesn't make a difference in the treatment

7.4 Conversion

Now we show that tagging objects with other sorts is a special case of yet another construct, known as conversion/coercion in the imperative programming world. This notion includes the idea that if we want to *compile* an imperative language without the assumption, that all data is represented as terms, then in general *it will be necessary* to change the representation as well, which is traditionally done by inserting functions into the parse tree.

Besides this rather ugly low-level detail, there are in the author's opinion two theoretical reasons to make the conversion an explicit part of the language description:

- If subsorts represent an embedding, the corresponding homomorphisms may be neglected by convention in mathematical texts – but not in a precise description of a program.
- A conversion "forgets" part of the information about a value, for example the knowledge that an integer was an integer before it was embedded to the reals – and this should be explicit somewhere.[16]

A programming language may want to (but few actually do) provide a declaration:

```
conversion s -> t by function [ if conditions  ].
```

where s and t are types and `function` is the unique name of the applicable function. In the case of re-tagging, this is simply the generic identity function.

Of course, s could be recursively nested, and `conditions` can give further requirements, which to the best of the author's knowledge is found nowhere else in this generality.

It is straightforward to see that L_{TE}'s expression for the above declaration is

$$x : s \to \texttt{function}(x) : t \quad \Big| \quad \begin{array}{l} \texttt{conditions} \\ \textit{unrolled} \end{array}$$

where *unrolled* stands for further conditions contained in the nested s.

Most notably, the `conditions` can involve statements about structural conversion:

```
conversion vector(s) -> vector(t) by (lambda(x) map(f, x))
        if s -> t by f.
```

[16] Section 8.2 makes use of this "forgetful operation" again.

One example in OTTER is

```
% x : nat -> nat_to_int(x) : int
 -rel(o$(_x,_t), o$(_x1, nat))
   | rel(o$(_x,_t), o$(n2i(_x1), int)).

% x : int -> int_to_real(x) : real
 -rel(o$(_x,_t), o$(_x1, int))
   | rel(o$(_x,_t), o$(i2r(_x1), real)).
```

Together with a simple overloaded function `neg` `int -> int` and `real -> real`

```
% neg(x) -> neg_r(x1) : real
%   where x -> x1 : real
 -rel(_a, o$(_x, real)) |
 rel(t1$(neg, _a), o$(neg_r(_x), real)).

% neg(x) -> neg_i(x1) : int
%   where x -> x1 : int
 -rel(_a, o$(_x, int)) |
 rel(t1$(neg, _a), o$(neg_i(_x), int)).
```

These clauses can of course be written more nicely as

$$x : \text{nat} \to \text{n2i}(x) : \text{int}$$
$$x : \text{int} \to \text{i2r}(x) : \text{real}$$
$$:_T \text{neg}(a) \to \text{neg\_i}(a) : \text{int} \quad | \quad a \to x : \text{int}$$
$$:_T \text{neg}(a) \to \text{neg\_r}(a) : \text{real} \quad | \quad a \to x : \text{real}$$

We can ask the question about all the interpretations of the negation of a natural, i.e. we have to judge

$$:_T \text{neg}(c : \text{nat}) \to x : t$$

which is equivalent to

```
 -rel(t1$(neg, o$(c,nat)), o$(_x,_t))| $ans(_x,_t).
```

This yields the proofs:

```
----- PROOF ----- 13 [binary,12.1,5.1] $ans(neg_i(n2i(c)),int).
----- PROOF ----- 16 [binary,15.1,5.1] $ans(neg_r(i2r(n2i(c))),real).
----- PROOF ----- 20 [hyper,12,5] $ans(neg_i(n2i(c)),int).
----- PROOF ----- 22 [hyper,15,5] $ans(neg_r(i2r(n2i(c))),real).
```

This means that we can get either a final result of type `int` or `real` and furthermore the parse-tree is annotated with the conversion functions, such that it can be directly compiled.

7.5 Higher-order functions

Functions as values are useful in many situations, especially for computing in structures such as rings or ordered sets. In this section we show that functional values are no special case: They can even be converted ! (Note that they have one co- and one contravariant position).

Unfortunately, we will have to state conversion rules for every arity we want to use, but since the automatic translation from the programming language handles this, it should not be too cumbersome. To get the full benefits from higher-order functions, we will need to make use of closures, which construct a function from a term. The `apply` operation must be changed to handle the substitution. [17] For a unary function the rule would be ($\rightarrow$ is the function space constructor):

$$f : (a \rightarrow b) \rightarrow (\lambda \mathrm{v}.w) : (a' \rightarrow b') \quad \Big| \quad \begin{array}{l} \mathrm{v} : a' \rightarrow v' : a \\ f(v') : b \rightarrow w : b' \end{array}$$

The idea behind the construction is:

1. Convert the variable v[18], neglecting the fact that it is a variable, from a' to a (contravariantly)
2. Apply f to the resulting term and convert this, which is now of type b, to b' (covariantly)
3. Close the resulting term to yield a function.

In OTTER the approach looks like this

```
%% lambda abstraction is done in de Bruijn notation
  -rel(o$(v(0), _a1),   o$(_v, _a)) |
  -rel(o$(apply(_f, _v), _b), o$(_f1, _b1)) |
  rel(o$(_f, func(_a,_b)), o$( lambda(_f1), func(_a1, _b1))).
```

To get hold of functions as values, the declared functions must be made available via rules. Suppose a function is declared as:

```
function f(x:s):t = ...
```

[17] The de Bruijn notation is one way out of the necessity of generating fresh variables within the calculus, but adjusting variable numbers by one is necessary during functional abstraction and β-reduction.

[18] v is a variable of the programming language, not of L_{TE}. Therefore it is type-set in roman font and treated as a constant in the inferences.

If F is the unique, mangled name of f, then the rule to be inserted to the database is

$$:_T f \to GLOBAL(F) : (s \to t)$$

Whenever an f appears in a term, it can now be interpreted as a reference to the function. The wrapper `GLOBAL` is used to distinguish the unique name F, which in the final code ends up to be a call to F, from a `CLOSURE`, which is possibly a temporary object and cannot be called directly (for the handling of closures in stack machines see e.g. [29, p.221]). In the examples, this distinction is not carried out firmly for the sake of readability:

```
rel(t0$(f), o$(lambda(f_nat_int(v(0))), func(nat,int), no_c)).
```

That function conversion works can be seen in the following examples. With the knowledge (conversion rules) about `int`, `real`, `nat` as above and the function f, which we leave as a symbol for this example, we can ask the question:

```
% f : func(int,int) ->? f' : func(nat,real)
 -rel(o$(f, func(int,int)), o$(_X, func(nat,real)))
 | $ans(_X).
```

The proof yields the expected behaviour:

```
----- PROOF ----- 35 [hyper,25,1]
$ans(lambda(i2r(apply(f,n2i(v(0)))))).
```

The new function is

$$\text{i2r} \circ f \circ \text{n2i}$$

written as a closure in de Bruijn notation.

> Note that in the special case of subsorts, the identity functions need not even be mentioned in the `nat->int` and `int->real` rules, and therefore they would not show up in the final result, which would be `$ans(lambda(apply(f,v(0))))$` The `apply` could be optimized away by η-reduction.

We can also go back to the list example: We can convert a whole list by `map`ping the conversion functions for the elements to the lists, which can be stated as:

$$l : \text{list}(\alpha) \to \text{map}((\lambda v.c), l) : \text{list}(\beta) \quad | \quad v : \alpha \to c : \beta$$

```
%% a conversion : list<alpha> -> list<beta> if alpha->beta
%% can be done by mapping of the conversion function
  -rel(o$(v(0), _alpha), o$(_conv, _beta)) |
  rel(o$(_x, list(_alpha)), o$(map(lambda(_conv), _x), list(_beta))).
```

The technique is again to convert the de Bruijn variable $v(0)$ and get a term, which we close and `map`. The map would be stated by the user in the body of the conversion declaration and solves the problem of structural coercion, i.e. how in general to construct the conversion function.

Now we can compute the example:

```
%% Check that list<nat> can be converted to list<real>
%% x : list(nat) -> X : list(real)
  -rel(o$(l, list(nat)), o$(_ll, list(real)))
  | $ans(_ll).
```

with the solution

```
----- PROOF ----- 43 [neg_hyper,21,1]
$ans(map(lambda(i2r(n2i(v(0)))),l)).
```

The new list is built by mapping i2r $\circ$ n2i to the list.

In [24, p. 127] Reynolds makes the statement

> At first sight, functions that accept polymorphic functions seem exotic beasts of dubious utility. But the work of a number of researchers sugests (sic!) that such functions may be the key to a novel programming style.

Combining the features

- function-conversion (with co/contravariance)
- structural coercion (of lists)
- generic functions with instantiation

we get the following, quite sophisticated, example where we want to convert a *generic* function `head`, which we have obtained from a global function declaration, to a *specialized* version:

```
% head : func(list(_alpha), _alpha)  ->? F : func(list(int), real)
 -rel(o$(head, func(list(_alpha), _alpha)),
      o$(_f, func(list(int), real )))
  | $ans(_f).
```

Note that in most systems, this does not work, because apparently α has to be two different types. It works in the $\rightarrow$ calculus, because we have strictly sequentialized the unification and inference relation.

As can be expected, there are *two* structurally different solutions: We can either convert the list and then apply the generic function, or we can apply the function and then convert the result. In any case, `head` is instatiated with both α as the same type, such that the requirement from its declaration is not violated:

```
----- PROOF ----- 38 [hyper,25,1]
$ans(lambda(apply(head,map(lambda(i2r(v(0))),v(0))))).

----- PROOF ----- 54 [binary,53.1,1.1]
$ans(lambda(i2r(apply(head,v(0))))).
```

If the function is not a closure, i.e. it is a symbol defined by another rule, the apply rule (for binary functions) can be readily stated as:

$$:_T \text{APPLY}(f, x, y) \rightarrow r : t \quad \Big| \quad \begin{array}{ll} f \rightarrow f' : ((a, b) \rightarrow c) & (*) \\ x \rightarrow x' : a & \\ y \rightarrow y' : b & \\ :_T f'(x' : a, y' : b) \rightarrow r : t & \end{array}$$

Note that in line $(*)$ we make an assumption, which is satisfied by the way we gave access to global functions: The "value" part of the function object is a unique name for the function, usable in terms.

This structure has been derived from the OTTER input used in the next example

```
-rel(_f, o$(_f1, func2(_t1,_t2,_t3))) |
-rel(_x, o$(_x1, _t1)) |
-rel(_y, o$(_y1, _t2)) |
%% force eval for transitivity
-rel(t2$(_f1, o$(_x1, _t1), o$(_y1, _t2)), o$(_ret, _tret)) |
rel(t3$(apply, _f, _x, _y), o$(_ret, _tret) ).
```

It conceptually proceeds in these steps (line-wise)

1. Evaluate the function value
2. Evaluate the first argument
3. Evaluate the second argument

4. Evaluate the function (which is supposed to be a function symbol)

5. Return the value

If the function is:

```
rel(t0$(plus), o$(plus_i_i_i, func2(int,int,int))).
```

The the evaluation is:

```
-rel(t3$(apply, t0$(plus), t0$(3),t0$(4)),
     o$(_x, _t))
       | $ans(_x,_t).
```

and yields

```
----- PROOF ----- 51 [hyper,39,1,1,demod]
$ans(7,int).
```

We can be slightly more efficient when using the knowledge about our [trans] rule, which OTTER does not have. Strictly speaking, the inference rule could be stated as

$$:_T \text{APPLY}(f, x, y) \to :_T f'(x' : a, y' : b) \quad \Big| \quad \begin{array}{l} f \to f' : ((a, b) \to c) \\ x \to x' : a \\ y \to y' : b \end{array}$$

The difference is that we do not *within the rule* evaluate the term

$$:_T f'(x' : a, y' : b)$$

but do the type checking only and rely on the [trans] rule to proceed in the inference with the derivation

$$:_T f'(x' : a, y' : b) \to r : t$$

by itself.

7.6 Object-Oriented Programming

Having higher-order functions, the "records of functions" model can be directly translated to L_{TE}. A few technicalities need to be discussed:

- An object consists of a tuple $(d, f_1, \ldots, f_n)$ where d is the accumulated data and the f_i are functions. This organization yields essentially the same structure as the C^{++} virtual table-approach.

- Base-class conversion simply copies the derived classes data and those operations which also apply to the base class. This is where the co-/contravariance behaviour comes up.
- For method dispatching, we introduce a special symbol

  $$\texttt{DISP}(f, o, x_1, \ldots, x_n).$$

  f is a symbolic name (string), which is mapped to a slot f_i via L_{TE} rules checking the effective type of the object to dispatch.
- The parser must basically only convert the method-dispatch syntax. `x.f(y);` to `DISP(f,x,y)` to make it accessible to the $\rightarrow$ system.

7.7 Integrating Structures

Now that we have higher-order functions and type classes, we can easily introduce mathematical structures, where the carrier-sets, special constants, and operations are simply parameters to a type class.

Most notably, there is no difference in treating the third or second component of a `o$()` encoding, or in the $\rightarrow$-system, there is no difference between elements of paths. Therefore, we can get type classes with co- and contravariant parameter positions, just as we did before for functions. One example is: Treat the integers as a monoid. This can certainly done in two distinct ways (see section 7.12) one of which is:

$$:_t \text{int} \rightarrow :_t \text{int} : \text{Monoid}(+_{\text{int}} : ((\text{int}, \text{int}) \rightarrow \text{int}), 0_{\text{int}} : \text{int})$$

In a programming language, most of the information is redundant: The value and type components are not changed. Therefore it makes sense to devise a syntax construct as:

```
structure of type  t  is  C.
```

For the above example, depending on whether we want to allow overloaded identifiers in this special situation, we would write:

```
structure of type int is Monoid(+,0).
```

Since non-determinism is present in the inference process, several statements about a single type can be made independently from one another.

In OTTER this assertion looks as follows

```
%% say that the integers are a monoid with +,0
 -rel(o$(_x, _t, _c1), o$(no_v, int, _c)) |
 rel(o$(_x, _t, _c1),
        o$(no_v, int, MON(o$(plus_i_i_i, func2(int,int,int), no_c),
                          o$(0,int,no_c)))).
```

A question with it's proof is of course:

```
  -rel(o$(no_v,int, no_c), o$(no_v, _t, MON(_f, _n))) | $ans(_f,_n).
```

The proof is:

```
----- PROOF ----- 40 [binary,39.1,1.1]
$ans(o$(plus_i_i_i,func2(int,int,int),no_c),o$(0,int,no_c)).
```

Although such a question by itself is useless, note that a term

```
-rel(_T, o$(no_v, _t, MON(_f, _n)))
```

could well be generated from a function interface such as

```
function fold(x:list(T : Monoid(op, neutral))) -> T
```

To play a bit with the expressiveness at this point, the designer might want to make very clear that `fold` traditionally expects an explicit argument, we could have written:

```
function fold(x:list(T), T : Monoid(op, neutral)) -> T
```

where the second parameter is a type parameter stating what the list should be regarded as.

7.8 Data Construction and Destruction

Since OTTER knows how to match terms, it can easily deal with value construction and destruction, for example the basic list functions are readily defined:

```
%% Lists can be constructed using cons and null
%% null -> null : list(_alpha)
 rel(t0$(null), o$(null, list(_alpha), no_c)).
%% cons(a,b) -> co(A, B) : list(alpha)
%%   where a -> A : alpha
%%         b -> B : list(alpha)
 -rel(_a, o$(_a1, _alpha, _c1)) |
```

```
 -rel(_b, o$(_b1, list(_alpha), _c2)) |
 rel(t2$(cons, _a, _b), o$(co(_a1, _b1), list(_alpha), no_c)).

%% Deconstruction using is_null, car, cdr
 -rel(_l, o$(co(_car, _cdr), list(_alpha), _c)) |
 rel(t1$(car, _l), o$(_car, _alpha, no_c)).

 -rel(_l, o$(co(_car, _cdr), list(_alpha), _c)) |
 rel(t1$(cdr, _l), o$(_cdr, list(_alpha), _c)).

 -rel(_l, o$(null, list(_alpha), _c)) |
 rel(t1$(is_null, _l), o$(true, bool, no_c)).

 -rel(_l, o$(co(_x,_y), list(_alpha), _c)) |
 rel(t1$(is_null, _l), o$(false, bool, no_c)).
```

Note that the value component of an object contains *untyped*, internal constructors. Again in the area of stack machines [29, p.221], the internal representation of a constructor on the heap, when applied to the appropriate arguments, is

$$(i, x_1, \ldots, x_n)$$

where i is a numerical tag to dynamically identify the data structure for purposes of pattern matching.

That the approach works together with conversion can be seen in trying to introduce integers into a list of reals (the tail is given here explicitly and the other `t0$` evaluate only to integer constants):

```
   -rel(t2$(cons, t0$(3),
               t2$(cons, t0$(2),
                         t2$(cons, o$(1, real, no_c),
                                   t0$(null)))),
      o$(_l, _t, _c))
 | $ans(_l, _t).

%%-------- PROOF -------- 3288 [hyper,3280,16] // 3.5 sec
%%$ans(co(i2r(3),co(i2r(2),co(1,null))),list(real)).
```

All but the last value need to be converted.

7.9 Calling and Computing with Generic Functions

A lot of the text has been aimed towards compilation. Now we want to justify the name L_{TE} — A language for typed execution. We evaluate the recursive `length` function directly.

The function in a programming language would be written as

```
function length(l:list(alpha)) : int =
       (if (null? L)
           0
           (+ 1 (length (cdr L))))
```

When translated to OTTER, as usual the arguments are type checked first, then the body is evaluated and then the result is returned as the consequence of the the inference rule:

```
%% Eval argument
 -rel(_l, o$(_ll, list(_alpha), _c)) |
%% Eval body
 -rel(t3$(if, t1$(is_null, o$(_ll, list(_alpha), _c)),
              o$(0, int, no_c),
              t2$(plus, t0$(1),
                        t1$(length, t1$(cdr, o$(_ll,
                                                list(_alpha),
                                                _c))))),
      o$(_le, int, no_c)) |
 %% return result
 rel(t1$(length, _l), o$(_le, int, no_c)).
```

Please note the the body does *not* contain any annotation except the usual number of arguments and could have been produced directly by a parser !

The question was

```
   -rel(t1$(length,
                  t2$(cons, t0$(2),
                            t2$(cons, t0$(3),
                                      t2$(cons, t0$(4),
                                                t0$(null))))),
         o$(_l, int, _c))
  | $ans(_l).
```

and again could have been syntactically generated.

Of course, we have to define how we evaluate the `if-then-else` construction:

```
%% (if x y z) -> y
%%     where x -> true : bool
 -rel(_I, o$(true, bool, _c1)) |
 -rel(_TH, o$(_x, _t, _c)) |
 rel(t3$(if, _I, _TH, _E), o$(_x, _t, _c)).
%% (if x y z) -> z
%%     where x -> false : bool
 -rel(_I, o$(false, bool, _c1)) |
 -rel(_E, o$(_x, _t, _c)) |
 rel(t3$(if, _I, _TH, _E), o$(_x, _t, _c)).
```

Some remarks seem appropriate:

- Obviously the boolean predicate will be evaluated more than once, but this doesn't matter in a purely functional context, about which we have been talking so far and which seems most appropriate for a high-level treatment.
- The `then` and `else` branches might be (partially) evaluated, before the boolean decision value is known. This doesn't lead to real conflicts, such as segmentation violations here, because inference is silently stopped, i.e. the clause is discarded from the set of support, when no further resolution is possible.
- We never run into endless loops because of breadth-first search, but it can be very inefficient.
- If we could direct OTTER's search and insist on proving the first goal before any others, we dispose of both
  - the inefficiency
  - the "instability"

Despite these remarks, OTTER found a proof

```
----- PROOF ----- 10538 [binary,10537.1,1.1]
$ans(3).
```

It turns out that when `length` is given an evaluated argument, the inference is much faster:

```
-rel(t1$(length, o$(co(4,co(3,co(2,co(1, null)))),
                    list(int), no_c)),
```

```
         o$(_l, int, _c))
| $ans(_l).
%-------- PROOF -------- 468 [neg_hyper,461,1]
%$ans(4).
```

Again, the remark seems in place, that we have to explicitly force the evaluation of the body within the rule only because OTTER does not know about the transitivity of $\rightarrow$. In L_{TE} the formulation could have been done slightly more elegant and also closer to the intention of the function:

$$:_T \mathrm{length}(l) \rightarrow :_T (\texttt{if}, (\texttt{null?}\ \texttt{l}'), \texttt{0}, (\texttt{1} + (\texttt{length}(\texttt{cdr}\ \texttt{l}')))) : \mathrm{int}$$
$$\mid \quad l \rightarrow l' : \mathrm{List}(\alpha)$$

The only check, before the *length* term is replaced by the body of the function, is the type-check of the argument. The return type is known from a previous check of the body, so we can use untyped reduction for the body.

7.10 Assignment Operations

ML has introduced the notion of a `ref` type, which serves as a wrapper around values to make them assignable. A similar system can be used here,too: We start out with a value of type `LOC(alpha)` where `alpha` is any type. Such values can be passed around just like any other object in our calculus.

When needed, they can be automatically dereferenced via a simple rule

$$x : \mathrm{LOC}(\alpha) \rightarrow \mathrm{DEREF}(x) : \alpha$$

```
% x : LOC(alpha)  -> deref(x) : alpha
  -rel(_X, o$(_x, LOC(_alpha), _c)) |
   rel(_X, o$(deref(_x), _alpha, _c))
```

providing the compiler again automatically with the *full* information, i.e. tree annotation, which is necessary for direct code generation.

Since the value is wrapped up in `LOC`, none of the other value inference rules will apply, as for example conversions. This yields the desired semantics: A reference parameter to a function (i.e. an `inout` parameter) has *invariant* behaviour—it is neither co- nor contravariant.

One question is how to obtain such references. Most easily, we can make a term referring to a variable yield a location. The declaration

```
var x : t .
```

must be translated by the parser to

$$:_T \texttt{x} \rightarrow \mathrm{mangled\_x} : \mathrm{LOC}(\texttt{t})$$

7.11 Sequencing

Once variables are accessible, sequencing of operations makes sense and can easily be done via a special term `SEQ(x,y)` which assumes that x is evaluated before y (if we restrict ourselves to compilation, not evaluation, then again this doesn't matter, since variable-modifications won't be *done* anyway).

7.12 Directing the Interpretation

Often it will be necessary to "push" the inference process into the right direction, for example when ambiguities arise. That we don't need any extra mechanism on a meta-level can be seen by the following simply `AS` rule:

$$\mathrm{AS}(X, Y) \to Y \quad | \quad X \to Y$$

```
-rel(_X, o$(_y, _t, _c)) |
 rel(AS(_X, o$(_y, _t, _c)).
```

We have to use a general `_X` instead of `o$(_x,_S, _D)` because `_X` may well contain a term `tn$` still to be evaluated.

The idea is that many ways might lead to the fulfillment of the precondition, but only the one which is given by the user is passed on to the next inference step. Since the second argument can contain variables, we may leave parts of the specification open or have them retrieved by the system for convenience, although we could also give them precisely with more effort. This corresponds to "loose" mathematical thinking: Treating the integers "as a ring" with the usual understanding of how and why the integers form a ring.

7.13 User Definable Types

High-level languages feature a mechanism for defining abstract data types, i.e. (1) a type name, (2) a (private) representation, (3) a collection of functions operating on the representation.

In L_{TE}, type names need not be introduced explicitly, so there is no real need to express that part of a declaration. However, it might be good for reasons of efficiency in the inference process. Suppose a datatype is declared as

```
declare data t((+)s:b) = ...
```

The new data type t has one (type) parameter s with bound b and covariant behaviour. Whenever an interface of a function uses this data type t, it must explicitly repeat the bound of s. This is particularly useful in the case of generic programming, because it requires and allows a precise statement of local preconditions. The inference process however, cannot make use of the knowledge, that no type t can be build without the parameter having bound b. This invariant is therefore checked over and over again, even if the bound is the same as in the declaration!

The solution is again to have a unique encapsulation of t, say $T(s)$ where no bounds are present. A correct type expression $t(s')$ where s' fulfills the bound s, can be changed to this internal format. In the interface of a function, one would use the keyword `usual` to indicate that no special bounds are needed.

In rules of L_{TE} this looks as follows

$$:_T\ t(s) \to :_t\ T(s) \quad | \quad s \to b$$
$$:_T\ T(s) \to :_t\ t(s)$$

```
function f(x: usual t(s)) = ...
```

would be expressed as

$$:_T\ f(x) \to \ldots \quad | \quad x \to T(\mathrm{s})$$

The semantics without using T is retained in connection with the above conversion rules, so indeed internal names of types are only a technique of "wrapping up invariants".

Concerning the separation of representation and usage, ADA has developed a sophisticated module system, and the the concept of encapsulation seems to be adaptable to our situation without too much effort. Especially, a module (or even a single declaration) might incorporate a private area, within which the values of type t can be treated as equivalent to their representation (a conversion function). Everywhere else an encapsulated value of type t must be modified using the functions within the private area.

8 Extensions and Further Thoughts

8.1 Towards Type Inference

The Hindley/Milner system for type inference for expressions *without* requiring the variables in the leaves to be declared. It has not yet been tried to translate this process, i.e. the inference rules found in [18], to L_{TE}.

The reason why we think that it could be possible is the following: The overload resolution rule proceeds up the parse-tree and annotates it with expressions. However, unification makes it possible to instantiate a variable in a node *after* the node has been visited. If we tag all leaves to have distinct type variables as types, then these will be instantiated *as soon as* a condition upon the structure of their type is found. Together with a general rule for λ-application, which forces the first parameter to have a function type and unifies the argument with the argument position of this function, the inference process could well simulate algorithm $\mathcal{W}$.

8.2 Solving a Longstanding Problem

In July '98 the author has posed the following challenge, which seems harder than Stepanov's `min/max` test, to generic programming languages :

> Given two type classes, `Sequence` and `OrderedSequence` (or `Sequence{ordered}` with attributes). Obviously they are related in most respects, for example for printing, iteration,... However, they don't share the `append` operation.
>
> Can their relation be captured in the language ?

The crucial, and in itself most discouraging observation is, that they are not related by the generally available *substitutability* notion.

If `append` is a function, then it certainly cannot be asserted: For example appending element 1 at the end of the `OrderedList` (which is an instance of the `OrderedSequence`) $[2, 3, 4]$ certainly violates the definition of `OrderedList`

But making `append` an operation, which comes along with the class, does not help either, because still it could applied to a `OrderedList` value and destroy the invariant.

The observation leading to a solution can be stated in two ways:

1. The assertion about a type *class* indirectly and *implicitly* (which we wanted to avoid strictly !) influences the *type* of a value.

2. The situation can be referred back to the circle-ellipse dilemma, where an operation `stretch` changed the `self` type, which is by definition not possible in OOP languages.

The way we solved the circle-ellipse dilemma was to replace the notion of *substitutability* by a *forgetful operation*, which we identified with the well-known concept of embedding or *conversion*.

At this point, all we have to do is introduce *higher order conversions*. In the language L_{TE}, they can be expressed as:

$$x : t : \mathrm{OrderedSequence} \to x : \mathrm{nonOrdered(t)} : \mathrm{Sequence}$$

The `nonOrdered` is a function *from types to types* which we had been thinking about in section 3.3. We said there, that this function may contain even `while` loops, and be Turin-complete on the domain of types (instead of other values, which in L_{TE} is an artificial distinction).

There is basically two ways to implement the required function:

1. For every type, which is an `OrderedSequence`, the function returns the corresponding type without the invariant. Note that this might even make a more general scheme necessary, if representation is changed during the process, e.g. from red-black-trees to lists:

   $$x : t : \mathrm{OrderedSequence} \to \mathrm{nonOrdV(x : t)} : \mathrm{nonOrdT(t)} : \mathrm{Sequence}$$

   The first function needs needs the type as well to determine, which changes will be needed.

2. If the invariants are stated in the form of symbolic attributes, the `nonOrdered` function would fetch the list of attributes from the type, loop over it, and kill the undesired attributes.

L_{TE} allows us to use either version and it is a matter of taste, which one to prefer.

Note that the forgetful statement is valid *regardless* of which algorithms are declared, and therefore the idea can be considered as general as the substitution principle.

8.3 The Program Structure

Of course what has been given so far does not yield a programming language. What is in particular missing is the derivation of the context , from the syntactical program structure.

What we are aiming at is a set of modules, which consist of definitions of functions, some of which may be value constructors, but for a first sketch we treat them equal. The interfaces of the function, as a $\to$-rule, will be automatically derived from the definition and bound to the body by means of a compiler-generated, program-wide unique name. This solves the linking problem imminent in C/C^{++}.

8.3.1 Compiling Generic Functions

For the compilation process, in which the information given in the declaration of a function together with the declaration of local variables must be converted, such that inferences in the compilation of the body can be made. In particular this means that

- all variables, including type variables, are regarded as constants
- local variables are given just as input-value parameters are, making their location open for read/write via the `LOC` types.

The interface-translation discussed so far yields a structure like:

```
% parameters = preconditions
-rel( ... ) |
-rel( ... ) |
-rel( ... ) |
% body = evaluation
 rel( ... ).
```

When compiling a function, the single declarations of input parameters and local variables will be sequentially inserted into the environment. For example, a local variable would be declared as:

```
var x : int;
```

which would be translated, recalling that the name may be overloaded and denotes a memory location, as:

```
x -> (&local_x) : LOC(int)
```

where `&local_x` stands for the address within the stack frame. [19]

The body of the function must of course yield a value of a type according to the return type, which makes for a clause similar to the initial judgments we have given in all our examples.

Overall, the OTTER-version of a function-compilation looks like this:

[19] Strictly speaking, there will be different `LOC` types, for example `LOC_STACK`, `LOC_GLOBAL`, ... according to the addressing schemes known on the (abstract) machine model, for which we compile. Accordingly, there will be different `DEREF` functions, which emit code for stack references, global references, ...

```
% parameters = declarations = assertions
rel( ... ) .
rel( ... ) .
rel( ... ) .
% body = evaluation : Show that the output interface is respected
 -rel( ... ).
```

As now we see the interface from the other side, i.e. from inside, the proof-obligations are reversed:

- We can *assume* that the input requirements are met and
- must *show* that the output is correctly produced.

Since the type variables can be arbitrarily instantiated, we cannot make any assumptions *except their declared minimal requirements* and treat them as constants, as we do with any other input parameters.

We call the two forms given above the *inner and outer translation* of an interface.

Two most interesting, yet not very profound observations in the theoretical direction arise immediately:

- Looking at the Horn-clauses, the inner translation of an interface is the *negation* of the outer translation.

  This can be well explained by the similarity between our $\rightarrow$ relation and the logical implication $\Rightarrow$ with respect to the property that there is a number of preconditions, which must all be satisfied to yields one conclusion.

- Concerning the treatment of variables as constants, Milner [18, p. 362] has observed a similar property: Only the variables bound in function declarations (via `let`) (which he calls the *generic* variables of the expressions), are free for substitution, the others are treated as constants!

8.3.2 A Module System

How can a pre-compilation of a generic function be accomplished ? Obviously, one can compile the body of a generic function with respect to the inner translation of the interface. Respecting the remarks in section 3.2 about parameters of algorithms, a module on disk will consists of two blocks:

1. The outer translations of the interface of all functions are at the beginning. When a module is imported, they are simply merged with the existing rules in the , of another module.

   Each of the rules has the form:

   ```
   o_name(X) -> u_name(X,Y) : type
       where conditions
   ```

   The overloaded name is replaced by an expression in the unique name and is asserted to have a type (because the real return value is not known at compile time).

2. The bodies of the functions are placed in the module as templates ready for *untyped* substitution and code generation[20].

8.4 Run-time Well-Typing

Inductively, since every function body satisfies the inner translation of its interface, and there are no interfaces without uniquely determined bodies, nothing can go wrong in the calling sequence of a function.

The induction base for this proof are of course the built-in primitive types, which must be stated as:

```
o_name(X) -> BUILT_IN(name, X) : type
```

hoping that the compiler-constructor knows what he's doing (but this can be verified).

Via this little scheme, we could in principle, if only our programming language receives a rigorous treatment, have a result saying that computation can't go wrong, just as in Milner's "Theory of Polymorphism".

8.5 Symbolic Attributes as Restrictions

In [26] Schupp points out that it is very desirable to be able to modify declared types by further attributes locally. Instead of declaring a type *MonicPolynomial*, one would rather prefer to write $Polynomial\{monic\}$. Especially when many different attributes and subsets of these are needed, it is necessary to deal with these situations efficiently.

[20] One may want to re-examine the overload-resolution process when doing optimizations, because after substitution, the more specialized types in the body may give rise to a more specialized, and efficient resolved expression

However, during the discussion of attributes, we should keep in mind not to assign a *semantical* meaning to any of L_{TE}'s constructs. Rather, L_{TE} is designed such that we can easily reduce the semantical questions of a programming language to the offered inference mechanisms.

Recall that types are represented by terms, i.e. a type is described by a constructor, possibly with arguments. A type carries two pieces of information: First, it describes a set of values (on a symbolical level, not necessarily with representation). Second, the algebraic properties of this set are determined in terms of applicable functions since the type is used in overload resolution. In the translation to L_{TE}, the *types* form a class in the graph of simple objects, with possibly several parent classes, the "natural" class being *values*.

1. Although the present discussion is carried out in terms of types and values, it should be clear that it easily generalizes to any relationship in the L_{TE} graph.

2. Not every attribute is applicable to every type.

3. The meaning of an attribute may vary with the type it is applied to.

4. Attributes behave like set-theoretic *restrictions* to the value set in the sense of comprehensions

$$t\{a\} \equiv \{x \in t \mid a(x)\}$$

5. Therefore semantically attributes belong to a parent node in L_{TE}'s graph, which may not be unique. A further constraint on the applicability is imposed.

6. With this explanation, attributes represent *additional* information, which may be neglected in some cases.

7. Attributes are not only assertions, but also requirements, depending on whether they appear on the left-hand-side or right-hand-side of a judgment.

8. The value, to which the predicate applies, may not be directly accessible, as for example in the question, whether we can infer

$$\mathrm{Vector}(\mathrm{int}\{\mathrm{positive}\}) \to \mathrm{Vector}(\mathrm{int}\{\mathrm{nonZero}\})$$

9. Conceptually, attributes are sets. The simple unification used so far to solve the system of equations generated by **equ** (see section 5.4.3) does not have the capability to deal with such sets. Therefore, we will have to extend L_{TE}'s capabilities.

10. The order of attributes may be important, if the applicability of an attribute (and the corresponding predicate, which is a boolean function *with* a signature) depends on other attributes. This leads to type descriptions such as

    $$t\{A\}\{B\}\cdots\{Z\}$$

    where the $A, B, \ldots, Z$ are sets of attributes. Some of the attributes may be migrated to the left (if no dependencies arise) thus possibly leaving empty sets which can be deleted.

11. The problem with substitutability, which we pointed out in section 8.2 is *not* solved by introducing attributes. Neglecting an attribute is a truly forgetful operation, which is automatically included in the semantics of an attribute without requiring further L_{TE} rules.

From this list we conclude that further attention is necessary to avoid prematurely introducing rules which conflict with the concepts described so far. Specifically, the need for attributes can be seen as a motivation for examining forgetful operations systematically.

8.6 Verification Issues

One goal of the generic programming paradigm is to verify programs on a high level and to instantiate them correctly then later on. Since algorithms are verified in terms of e.g. first order predicate calculus, it would be desirable to connect our system to a verifier. This is in fact quite simple, at least from a bird's view perspective.

The initial idea is similar to the embedding of arbitrary predicates in section 5.5.2 : We *define* as equivalent a construct from L_{TE} and a construct from an arbitrary other language for judgments.

> Just like finding a homomorphism between algebraic structures, this step *reflects back* proof obligations of the other language to L_{TE}. The inferences made via $\rightarrow$ must respect the relations between the foreign objects and this is, besides the usual "no

program goes wrong", a *very strong, profound* and *most desirable* goal for the rigor of the semantic definition.

L_{TE} is designed for computation, for making decisions in a symbolic and efficient way, and on purpose does not carry any higher semantic meaning by itself, to allow for a later assignment as desirable. It has been constructed to carry out the inference steps needed, but not to understand why they are correct and with respect to what logical system.

We are convinced at this point, that TECTON is the ideal language for this purpose and believe that especially the effort taken in completing a standard `libtec` will make the combination easy. The reasons why the language concept fit so seamlessly with L_{TE}'s main ideas are the following:

- TECTON knows the concept of substitution and assigns the same meaning to it as L_{TE}.
- TECTON's requirements are intended to be handed on to a theorem prover directly.
- TECTON makes no implicit substitutions. The situation of finding proper instantiations for the parts of a concept can be dealt with analogously to the way we can compute the type parameters to algorithms and make them explicit (section 3.2).
- Inference rules in , and TECTON are related by the following diagram, which must commute for a properly defined language:

$$\begin{array}{ccc} O_1 & \rightarrow & O_2 \\ \| & & \| \\ C_1 & \overset{\text{refines}}{\Longrightarrow} & C_2 \end{array}$$

In a final system, one could have a compilation mode, which spills out precisely all the proof-obligations connected to the inference rules given in L_{TE}, i.e. the programming language, in terms of their TECTON equivalents. The substitutions in the form of `with` clauses are generated.

This yields a input text to the TECTON system, which can be checked *off-line* and once-and-for-all with as much computational power as needed, the compilation time for the program is not affected. Having that all the inferences in L_{TE} are legal and "certified" within the TECTON framework, we know that the resulting compiled program is not only correct with respect

to the inference rules given, but with respect to first-order predicate calculus or whatever machinery will be necessary.

Why we have put an emphasis on the `libtec`, is that all the example concepts and refinement-relations between them, are available from the beginning and can be (almost directly, i.e. without modification except syntax) used in a proper library of L_{TE} inferences, which in turn can be used in programs.

9 Conclusion

In this essay, we have treated the problem of designing a programming language from a novel perspective. Instead of giving rules, which we want the objects expressible in the language to obey, in a meta-language and implementing these meta-rules in a compiler one by one, we have given a language for typed execution L_{TE}, which provides essentially only one basic inference rule. The introduced relation $\rightarrow$ can be semi-decided using the well-known techniques of

- unification
- breadth-first search in an inference tree

and we conjecture that the *subset* of rules and judgments arising from the translation of programs, is decidable, given an application-specific search strategy. The following programming language constructs have been shown to be directly expressible in L_{TE} with their usual semantics:

- type checking with subtyping and type classes
- overload resolution with run-time decisions, i.e. dependent types
- partial specialization by constant folding, possibly with the same function bodies, which are later on compiled into the executable
- higher-order functions
- contravariant positions in both type- and type-class constructors
- automatic instantiation of type variables
- recursive checking of parameter-preconditions, including value-predicates and n-ary predicates
- explicitly controllable structural subtyping

- the "records of functions" object model

One of the remarkable points is that there is absolutely no difference in the complexity of *types* and *type classes*, making any artificial distinctions between these in the implementation superfluous.

We have presented a three step model of compilation:

1. Programming language
2. $\rightarrow$ calculus
3. Object code

Since L_{TE} can be "compiled", i.e. the results of the inference process can be fed directly to the code generator and the parser translating the programming language to L_{TE} can act on syntactical considerations only, we conclude that the above features could be effectively realized in a language along the lines of the system described, given that a decision procedure is implemented.

As extensions, which seem plausible to be realized, we have presented a module system, which includes the pre-compilation of generic functions to a point, where only syntactic substitutions are necessary to generate instances (linking process). They contain a set of L_{TE} inference rules, which are simply merged to an existing data base when the module is imported. The connection to the TECTON[19] system for having verified programs has been sketched.

10 Acknowledgments

The author wishes to express his gratitude towards Prof. R. Loos (University of Tübingen) for his understanding support and many helpful discussions, most notably about the TECTON language, during the work on this essay. Especially, the remarks given after reading several earlier versions have greatly helped to improve the presentation of the material.

Further thanks go to Prof. S. Schupp (Rensselaer Polytechnic Institute) for her support and motivating examples in summer '98 and for reviewing several short essays containing partial observations put together here.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996-1998.

[2] T.P. Baker. A One-Pass Algorithm for Overload Resolution in Ada. *ACM Transactions on Programming Languages*, 4(4):601–614, 1982.

[3] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 1985.

[4] Giuseppe Catsagna. Covariance and Contravariance – Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 1995.

[5] H. Comon, D. Lugiez, and Ph. Schnoebelen. A Rewrite-based Type Discipline for a Subset of Computer Algebra. *Journal of Symbolic Computation*, 11:349–368, 1991.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[7] H.-D. Ebbinghaus, H. Hermes, F. Hirzebruch, M. Koecher, K. Mainzer, A. Prestel, and R. Remmert. *Zahlen*. Springer Verlag Heidelberg, New York, Tokyo, 1983.

[8] Jean H. Gallier. *Logic for Computer Science – Foundations of Automatic Theorem Proving*. Harper & Row Publishers, 1986.

[9] Holger Gast. With Scheme from SuchThat to C^{++}. Studienarbeit, Universität Tübingen.

[10] Goguen and Meseguer. Introducing OBJ, 1993.

[11] Joseph Goguen and Răzvan Diaconsecu. An Oxford survey of order sorted algebra. *Math. Struct. in Comp. Science*, 1994.

[12] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 1996.

[13] Harper. Introduction to Standard ML. Technical report.

[14] H.Gast, S.Schupp, and R.Loos. Completing the Compilation of SuchThat v0.7. Technical report, Rensselaer Polytechnic Institute, 1997.

[15] Mark P. Jones. A theory of qualified types. *European Symposium on Programming*, 1992.

[16] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 1995.

[17] William W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL 94/6, Argonne National Laboratory, January 1994.

[18] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[19] David R. Musser. The Tecton Concept Description Language. `http://www-ca.informatik.uni-tuebingen.de/people/musser/gpseminar/tecton1.ps.gz`, 1998.

[20] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.

[21] John Peterson and Mark Jones. Implementing Type Classes.

[22] Martin Plümicke. Ordnungs-sortierte Algebren als Grundlage für Semantik and Typsystem einer algebraischen Spezifikationssprache, Diplomarbeit. Diplomarbeit, Universität Tübingen, 1993.

[23] Franco S. Preparata and Michael Ian Shamos. *Introduction to Computational Geometry.* Springer Verlag, 1985.

[24] John C. Reynolds. Three Approaches to Type Structure. In *Mathematical Foundations of Software Development.*

[25] J. A. Robinson. A Machine-Oriented Logic Nased on the Resolution Principle. *Journal of the ACM*, 1965.

[26] Sibylle Schupp. *Generic Programming Such That One can build an algebraic library.* PhD thesis, Universität Tübingen, 1995.

[27] Sibylle Schupp and Rüdiger Loos. SuchThat - Generic Programming Works.

[28] Robert Sedgewick. *Algorithms in C.* Addison-Wesley, 1992.

[29] Peter Thiemann. *Grundlagen der funktionalen Programmierung.* B.G. Teubner Stuttgart, 1994.

[30] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. *Principles of Programming Languages*, 1989.

[31] Andreas Weber. *Type Systems for Computer Algebra.* PhD thesis, University of Tübingen, 1993.

[32] Karsten Weihe. Generische Programmierung:Polymorphie jenseits von isolierten Objekten. Technical report, Fakultät für Mathematik und Informatik, Universität Konstanz, 1998.