The Type Inference and Coercion Facilities in the Scratchpad II Interpreter

Robert S. Sutor
Richard D. Jenks

Computer Algebra Group, Mathematical Sciences Department
IBM Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, New York 10598

Abstract

The Scratchpad II system is an abstract datatype programming language, a compiler for the language, a library of packages of polymorphic functions and parametrized abstract datatypes, and an interpreter that provides sophisticated type inference and coercion facilities. Although originally designed for the implementation of symbolic mathematical algorithms, Scratchpad II is a general purpose programming language. This paper discusses aspects of the implementation of the interpreter and how it attempts to provide a user friendly and relatively weakly typed front end for the strongly typed programming language.

Introduction

The Scratchpad II system is centered on a strongly typed abstract datatype programming language with multiple inheritance. The compiler for this language produces a library of packages of polymorphic functions and parametrized abstract datatypes. The system interpreter provides an interface to the compiler and library and supports a subset of the Scratchpad II language. The design of the interpreter includes extensive datatype inference and coercion facilities to ease the burden of working with a strongly typed language and a library containing over 350 modules.

Interpretation in Scratchpad II is unusual because:

- Scratchpad II has an infinite number of datatypes and packages,
- each generic operation generally has an infinite number of interpretations,
- datatypes and packages are instantiated dynamically in response to user input, and
- operation interpretation is done by pattern matching on modemaps.

Our philosophy in this design is that the

interpreter language = compiler language − constraints

This means that declarations are largely optional and that automatic datatype conversions may take place when they are deemed appropriate.

In Scratchpad II we use the term *conversion* to mean the transformation an object of one datatype into an object of another datatype. A special kind of conversion is *coercion*, where we further require that transformation is reasonable (usually in an algebraic or non-information-losing sense). Examples of coercions involve changing integers into rational numbers, rational numbers into (constant) polynomials with rational number coefficients, lists of lists into two dimensional arrays, etc..

Conversion is done through two operations, ***coerce*** and ***convert*** which are otherwise ordinary operations exported by the abstract datatypes of the system. The interpreter

is allowed to perform conversions named *coerce* whenever it determines they are necessary. Most coercions are reversible and their application does not lose information. An example of a conversion that is not a coercion is the transformation of a rational number into a floating point number. Having an automatic conversion from an exact to an inexact object is not acceptable.

All atomic expressions (as produced by the parser) have default datatypes. These are Boolean, Float, Integer, String, Symbol and List(None) (for an empty list). If these are assigned to declared variables or used as arguments to functions, coercions are often performed to create objects of other datatypes.

Coercion is combined with the *type resolve facility* to determine the result datatype of an expression presented to the interpreter. Datatypes T1 and T2 *resolve* to a datatype T3 if all objects belonging to T1 and T2 can be coerced to T3. In Scratchpad II, the resolve facility is driven by a rule system, by information about coercions and by the structure of the datatypes involved.

Overview of the Abstract Datatype System

It is not within the scope of this paper to fully discuss the Scratchpad II abstract datatype system. This section is an overview of those terms and concepts needed to describe the implementation of the interpreter, and other source are available that contain more detailed discussions [1] [3] [4] [5] [7] [8].

The Scratchpad II library consists of compiled code that creates "categories," "domains" and "packages." A *domain* is an datatype, a *package* is a collection of functions and a *category* is a collection of operation signatures and attributes, as shown below. These are created by special functions called *constructors*, which are generally parametrized.

Domains and packages implement the functions in categories. Categories serve to separate the contract from the implementation. However, default implementations can be given in categories by assuming the existence of other implemented operations. For example, a default implementation for binary "-" can be given by using unary "-" and binary "+".

Categories

A domain is an instance of an abstract datatype specified by one or more categories. Categories serve to group together domains with common properties (e.g., operations). A basic category in Scratchpad II is Set which has the definition:

```
Set(): Category == with
  "=":    ($,$) -> Boolean
  coerce: $     -> Expression
```

The syntax means the following. Set is a category constructor having no parameters. Lines 2 and 3 of the definition present *signatures* for the two operations that are exported by Set: an operation "=" that takes 2 elements of the set and returns a Boolean, and an operation *coerce* that takes an element of the set and returns a printable representation of it. The symbol "$" refers to the set itself. All domains which belong to Set contain "=" and *coerce* among their operations (though it is possible that they are not implemented).

Categories may be extended. A semigroup is a set that has an associative multiplication operation. Given such a multiplication, it is easy to define an exponentiation by positive integers (x**1 = x, x**2 = x*x, x**3 = x * x**2, ...). The definition of the SemiGroup constructor in Scratchpad II is

```
SemiGroup(): Category == Set with
  "*": ($,$) -> $
  "**": ($, PositiveInteger) -> $
  associative("*")
```

The last line of this definition presents an *attribute* declaring that multiplication is associative. Attributes declare properties of the operations or the datatype.

As an example, Integer belongs to SemiGroup which means that it should have everything from Set plus the listed multiplication and exponentiation operators. Multiplication is asserted to be associative and so this may be assumed for Integer. If a category C2 extends a category C1, any domain that belongs to C2 also belongs to C1.

Categories may be parametrized.

```
ListCategory(S: Set): Category == Set with
  first: $ -> S
  rest:  $ -> $
  null:  $ -> Boolean
  ...
```

presumably describes the operations and attributes that any abstract datatype that purports to act like a linked list should have. The parameter S can be any domain that belongs to Set. If we defined a special version of this

```
SemiGroupListCategory(S: SemiGroup): Category == Set
  with
    first: $ -> S
    rest:  $ -> $
    null:  $ -> Boolean
  ...
```

then S must have been explicitly declared to belong to SemiGroup. Category membership is by name: a domain having the operations and attributes of SemiGroup does not belong to this category unless it has been explicitly declared to belong to SemiGroup or one of its extensions.

Categories allow multiple inheritance.

```
FiniteList(S : Set): Category ==
    Join(Finite, ListCategory(S))
```

defines a category constructor that contains all operation

signatures and attributes from both the categories Finite and ListCategory(S).

Domains

Domains are created by domain constructors and are objects. They provide a representation for an instance of an abstract datatype, implement the operations of the categories to which they belong and presumably satisfy the attributes of the categories. The representation of a domain does not depend at all on the categories to which the domain belongs.

In the implementation of Integer, the following category definition is given and specifies the behavior of the domain:

```
Integer(): Join(UniqueFactorizationDomain,
  EuclideanDomain, OrderedRing, DifferentialRing)
    with
      oddp: $ -> Boolean
      random: () -> $
      numberOfDigits: ($,$) -> $
      abs: $ -> $
      canonicalUnitNormal
```

The category of Integer is not one named category but, instead, is composed of several pieces. The full category is that obtained by joining 4 named categories, 4 additional operations and 1 additional attribute. Note, though, that Integer belongs to each of the 4 named categories, *plus* their ancestors. In particular, Integer is a SemiGroup because it is a DifferentialRing, which is a Ring which is a SemiGroup. In fact, the Join creates several paths back to the category SemiGroup. Integer may be used anywhere a SemiGroup is required.

Domains may provide functions for objects other than those which they create. For example, RationalNumber implements a "/" for two Integer arguments.

Two domain constructors that are particularly useful in representing other domains are Record and Union. A record is similar to structures of the same name in many other languages: it has one or more fields accessed via selector names. The objects in the fields can be of any datatype. The constructor Union creates *discriminated* unions: an object of Union(Integer,String) belongs either to the Integer branch or the String branch of the union. The ***case*** function is used to test for membership in branches.

The constructor Mapping creates the datatypes of functions. Declaring `oddp : Mapping(Boolean,Integer)` states that ***oddp*** is function taking one integer argument are returning a boolean. A convenient alternate notation for this is `oddp: Integer -> Boolean`. Functions are first-class objects and can be passed to other functions.

The Scratchpad II language design completely separates the concrete implementation inheritance from the abstract specification inheritance. Domains can inherit implementations from their representations, but the external view of them is only determined by the categories to which they belong. The idea of *subtype* is languages such as Trellis/Owl[1] [6] is replaced in Scratchpad II by the two notions of category and domain.

In addition to providing an organizational hierarchy, categories allow Scratchpad II to have extra information about domains. Thus, for example, the compiler knows where operations will be located in domains and can compile efficient function calls. This information is part of a structure associated with each operation called a *modemap*.

Packages

A package is a special kind of domain in that it provides no new objects to the system (other than the package itself). It consists of a category listing operations and attributes and implementations of the operations. Because they may be parametrized, packages can be used to implement algorithms at their most natural level of abstraction. For example, a repeated-squaring algorithm may be contained in a package that is parametrized by a domain that is a SemiGroup.

Packages provide an excellent vehicle to implement both domain-specific and category-general algorithms. They can be used to implement algorithms that are not included in other domains for reasons of convenience or necessity. For example, the repeated squaring algorithm and the integration algorithms are implemented in packages. They also require extra work by the interpreter because argument datatypes give no simple clue to the possible location of a package function. For example, in the expression `2 + 9` the operation "+" is found in the domain Integer, which is the common datatype of the two arguments. However, the function ***solve*** in `solve(x**3 - 1,x)` is not present in either the domain Polynomial(Integer) of the first argument or the domain Symbol of the second. This, along with the last example of the previous section, demonstrates the need for a systematic search mechanism of the Scratchpad II library.

Modemaps

A *modemap* is a syntactic specification of an operation. It gives the name of the operation, source and target datatypes for its parameters, and the name of the datatype or package exporting the operation. For example, the modemap

oddp: Integer → Boolean from Integer

specifies an operation from type Integer which takes an integer argument and returns a boolean value. The "from"-clause indicates the *domain of implementation* of the operation (in this case, Integer).

When the datatype or package is parameterized, the conditions on their parameters appear in an "if"-clause for the modemap. For example, every domain created by domain constructor Matrix exports a trace operation:

[1] Trellis is a trademark of Digital Equipment Corporation.

trace: Matrix(R) → R if R has Ring from Matrix(R)

The domain constructor Matrix takes one parameter R which is required to belong to the category Ring.

Operations may also be conditionally exported by a domain constructor. For example, domains created by Matrix export a ***determinant*** operation only if the homogeneous multiplication "*" in the underlying domain is commutative, that is, R has ***commutative("*")***. The modemap for ***determinant*** is

determinant: Matrix(R) → R if R has Ring
and R has ***commutative("*")*** from Matrix(R)

Similarly, datatypes created by Matrix export an ***inverse*** operation only if their argument domain belongs to Field. Since the requirement "R has Field" implies "R has Ring," the modemap for ***inverse*** is

inverse: Matrix(R) → Union(R,"failed")
if R has Field from Matrix(R)

The collection of modemaps from all abstract datatypes and packages constitutes the *global modemap database* for Scratchpad II. To standardize the presentation of modemaps, arguments and other parameters are replaced by "pattern variables," here called *1, *2, etc.. The resulting form of the modemap for Matrix ***inverse*** in the modemap database is, for example,

inverse: *1 → Union(*1,"failed")
if *1 is Matrix(*2) and *2 has Field from *1

In addition to this general modemap database, each domain or package has a *local modemap database* for all its exported operations. Note that the pattern variables and "if"-clauses of the above global modemaps depend on the parameters to a domain or package constructor. For a domain or package itself, all parameters to the constructor are known. Thus local modemaps have no patterns. Predicates reduce to *true* or *false*. All modemaps in a local modemap database for a domain or package D therefore have the general form

f: S → T if *true* from D

Those having a predicate `if false` simply do not exist for a given domain or package. For example, Matrix(RationalNumber) will have an ***inverse*** operation whereas Matrix(Integer) will not, since RationalNumber is a field but Integer is not.

Interpretation

The role of the interpreter is to evaluate input expressions entered by the user. In addition to computations, these expressions may be declarations, function definitions or system commands. A value in Scratchpad II is described by a pair `<a,A>` where `a` is an object and A is its type. Evaluation of an operator-operand expression `f(x,y,...)`, with n ($n > 1$) arguments `x,y,...`, is done in a bottom-up manner. The interpreter will evaluate the arguments to produce a corresponding n-tuple of argument values `<a,A>`, `<b,B>`, ... At this point an attempt will be made to select an applicable modemap for *f*. If this modemap has a return type of T, *f* is applied to the (possible coerced) arguments to yield a result `t` and subsequent value pair `<t,T>`.

Modemap Selection

Given a function call `f(x,y,...)` and evaluated arguments `<a,A>`, `<b,B>`, ... as in the previous section, the interpreter tries to find an appropriate *f* to apply. The first attempt involves constructing a list of the domains A, B, ... of the evaluated arguments, plus the target type of the function call, if it exists,[2] plus those types contained in any of the argument types if they are constructed from Record, Union or Mapping. Duplicates are removed from the list and domains and packages in the list are searched for an applicable *f*. If a modemap is found of the form

f: (A,B, ...) → T from C

with arity n, for some T and C, the function is gotten from C and applied to `a,b,...` to yield a result *t* and subsequent value pair `<t,T>`.

This kind of search for an applicable *f* generalizes the idea of a *controlling object* in a function call [2] [6]. In Scratchpad II, though, the arguments in a function call need not give any hint to the actual location of the function to be applied. If there is no applicable modemap to be found among the argument domains of a function, a two-stage search for a suitable modemap is made in the global modemap database. A set M of candidate modemaps is constructed, each of the form:

f: (D,E, ...) → T if *p* from C

for some result type T and domain of implementation C, each of arity n and each generally containing pattern variables and predicates p. The set M is partitioned into two subsets: those modemaps coming from domains or packages whose names contain those of any of the arguments of the function, and those that do not. The modemaps in the first subset are examined for applicability and, if one is found, it is returned. Otherwise, those in the second subset are checked for applicability.

At first this partitioning may seem odd, but it reflects a naming convention for domains and packages in the Scratchpad II library. As an example, the constructor ListPackage1 takes one Set argument and implements several functions that could be but are not now implemented in the List domain. The constructor

[2] As an example of a target type, consider the expression `m : Matrix(Integer) := f(x,y,z)`. The target type of the function call is Matrix(Integer).

ListPackage2 takes two domain arguments, each belonging to the category Set. The functions in this package implement operations that require lists with two different element types. For example, the function

$$map: (S \rightarrow T, List(S)) \rightarrow List(T)$$

which maps a function from S to T across the elements of List(S), producing an object of List(T), is implemented in ListPackage2(S,T). Looking at the modemaps in the first partition of M reflects an extension of the search method that looks for applicable functions in the domains of the function arguments. In the future we anticipate using attributes in such associated packages to determine the first subset of the general candidate modemaps. That is, ListPackage2 will contain the attribute `associated("List")` in its category.

For our purpose here, it suffices to regard the predicate of a modemap as an conjunction of simple predicates of two kinds:

X is Y
X has Y

Predicates of the first kind state that pattern X matches only the explicit domain Y. If a modemap only has predicates of the first kind, patterns are replaced by the domain or package names to produce a new modemap free of patterns.

When the modemap has predicates of the second kind, substitutions are sought for the pattern variables such that the predicate is satisfied. Here Y is a category or an attribute. Pattern variables are given initial substitutions based on the value pairs of the arguments. Any pattern variable X for which there is a predicate "X is Y" has Y assigned as a *permanent* substitution. Other substitutions are labeled *tentative*. To satisfy the predicate, any or all the operations of *coercing*, *resolving* and *forcing* (see below) may be necessary. As above, if pattern variable X has permanent substitution Y, any argument value of f of type Z must be coerced to type Y.

The *resolve* of two types T1 and T2 will always succeed. It produces a third type T3 to which all objects of type T1 and T2 can be coerced. The type Any is used for T3 if a less general type cannot be found. Resolving is necessary to use a modemap with homogeneous arguments. For example, the modemap

"+": (*1,*1) → *1 if *1 has AbelianMonoid from *1

is used for addition in most algebraic domains in Scratchpad II. Given an expression 1 + (2/3), a bottom-up analysis will first produce <1,Integer> and <2/3,RationalNumber> as the values of the two operands of "+". The resolve of Integer and RationalNumber is defined to be RationalNumber. This causes the integer object 1 to be coerced to the rational number object 1/1. RationalNumber now matches *1 in the above modemap, the predicate evaluates to true, and the corresponding function is gotten from RationalNumber and applied to produce the result 5/3 and pair <5/3,RationalNumber>.

A *force* is an operation performed on one or more types to satisfy a predicate. For example, the predicate "*1 has Field" where *1 has a tentative substitution Integer results in the forcing of Integer to RationalNumber by the application of QuotientField, i.e. QuotientField(Integer) is equivalent to RationalNumber. As another example, x has default type Symbol. When appearing in a sum (e.g. x + 1), the modemap of x requires its argument to be from a domain which belongs to the category AbelianMonoid. As a result, Symbol is forced to Polynomial(Integer).

A list of all applicable modemaps is produced together with a list of required coercions on the source parameters. Duplicates and modemaps subsumed by others are discarded. Each modemap is assigned a cost based on the required coercions and the target type of the operation. By definition of the cost function, any modemap which directly matches the argument types (e.g. A=D, B=E, etc.) will have cheapest cost.

Ambiguity

Two remaining modemaps are said to be *ambiguous* unless there are coercions to and from the respective substitution datatypes for the pattern variables *1, *2, ... If there are no ambiguities, the cheapest modemap is selected and applied.

As a practical matter, users can always use a "package call" to avoid ambiguities, e.g. `x *$D y` and `foo$D(x,y)` direct the interpreter to apply the functions "*" and *foo* from the domain or package D. Package calling is the only way to identify uniquely functions of no arguments if they are ambiguous.

Modes

If the explicit conversion `p :: P` or `p :: P(□)` is given, p is converted to a polynomial. The datatype of coefficients may be any domain which satisfies the categorical requirements of the argument to the domain constructor Polynomial. The form P(□) is a *mode* specification rather than a *type* specification: the interpreter is free to choose what replaces the □.

A mode is a partial type specification in that zero or more arguments in a domain constructor call are replaced by □. The process of *merging* takes a type T1 and a mode M and determines type substitutions for the □'s in M to create a new type T2 to which T1 is coerceable. For example, the merger of RationalNumber and Polynomial(□) is Polynomial(RationalNumber). If the mode M has no □'s, it is a type and the merger will only succeed if T1 is coerceable to M. Thus merging a type and a mode may fail, unlike resolving two types. If M is sim-

ply □, then the result of the merger is T1.

Scratchpad II now only supports modes with at most one □, and that must be in the innermost constructor call position (e.g., List(Polynomial(□))). In our experience, this restriction has not seemed burdensome, though a future area of research might be the extension of the merging process to more general modes.

The Coerce Facility

Our goal in the design of the coerce facility is to have as much as possible controlled by modemap selection of compiled Scratchpad II functions. This allows the domain and package writer maximum control over the behavior of the interpreter and removes the requirement of having a system developer tune the interpreter for dealing with new datatypes. The coerce facility has several components.

Coerce by Function

The interpreter does modemap selection for an operation named *coerce* that has the appropriate argument and target types. This is the easiest way for a programmer to control the coerce facility. For example, the category Ring provides a modemap

coerce: Integer → R from R

where R is the ring being defined. Thus any ring has a coercion from Integer and, in fact, this operation has a default categorical definition.

Coercions can sometimes be defined in domains but are often defined in packages. A coercion of the form

coerce: Polynomial(QuotientField(R)) →
QuotientField(Polynomial(R))
if R has IntegralDomain

would typically be defined in a package parametrized by the domain R. On the other hand, it is not feasible to have so many explicit coercions being written and, in fact, there are general methods that will perform these kinds of coercions.

It is not now possible to define a coercion of the form

coerce: List(S) → List(T)

in the domain constructor List. This is because List is parametrized by only one set (S or T) and the modemap cannot, therefore, be part of the category of List. Such an explicit coercion can be provided in a package, but is, in fact, handled by the general mechanism described in the next section.

Coerce by Mapping

When the interpreter encounters a coercion of the form

D(T1) → D(T2)

where D is a domain constructor with a parameter (for reasons of exposition, we here omit the case of D having multiple parameters), it looks for a function

map: (T1 → T2, D(T1)) → D(T2)

that takes a function from T1 to T2 and an object of D(T1) and produces an object of D(T2). It then creates a function stub that coerces objects of T1 to those of T2 and passes it to *map*, along with the original argument.

Since the function *map* is part of the library of Scratchpad II compiled code, it allows the package writer to automatically provide an interpreter mechanism of "lifting" coercions from T1 to T2 to D(T1) and D(T2).

Coerce by Internal System Code

Some special cases of coercions are handled by internal system code rather than compiled Scratchpad II code. These typically involve polynomials where the variable ordering is changed or distributed across a tower of parametrized domains. It will eventually be moved into Scratchpad II code as the pattern matching facilities are improved.

Another case that is now handled internally involves rearrangement of a tower of parameterized domains. The domain constructor Gaussian creates domains with objects similar to the complex numbers in they contain real and imaginary parts. The coercion

Gaussian(Polynomial(RationalNumber)) →
QuotientField(Polynomial(Gaussian(Integer)))

is performed in the following steps. The type RationalNumber is changed into the equivalent type QuotientField(Integer) and the QuotientField is bubbled to the top of the original type to get QuotientField(Gaussian(Polynomial(Integer))). This new type and the old target now have the same top level constructor (QuotientField) and the coerce facility is called recursively on the underlying domains Gaussian(Polynomial(Integer)) and Polynomial (Gaussian(Integer)).

Coercion of Algebraic Constants

Several categories specify the existence of constants. For example, AbelianMonoid specifies the operation "+: $ -> $" and the constant 0 : $. If T1 and T2 each belong to a common category with specified constants and the object of T1 to be coerced is one of the constants, the corresponding constant in T2 may be extracted and returned.

Retraction

A retraction is a coercion of an object of a domain to a more specific (degenerate) domain. Unlike other forms of coercion where the target type is known, retraction involves examining an operand pair < t,T > to see if there

exists a degenerate form of T to which t can be coerced. For example, <7/1,RationalNumber> retracts to <7,Integer> and <1,Polynomial(Integer)> retracts to <1,Integer>. Retraction can occur multiple times, so arbitrarily long structures collapse to their simplest forms, e.g. <1,Polynomial(RationalNumber)> to <1, Integer>.

If no applicable modemap is found in modemap selection, retraction is done on the arguments and then selection is attempted again. Retraction is also attempted when a coercion is requested from a domain to its underlying domain. For example, if p is an element of Polynomial(Integer), the statement p :: I will cause the interpreter to try to retract p to an element of Integer.

For the most part, retraction can be accomplished by functions in the Scratchpad II library. The category RetractableWithUnderDomain(R) specifies two operations

```
retractable?: $ -> Boolean
retract:      $ -> R
```

In our example of polynomials above, the function *retractable?* would be called to see if the object was a constant polynomial. If the result was *true*, *retract* can be called to extract the constant.

Coercion Query

There are situations where one wishes to know ahead of time whether it is possible to coerce an object of type T1 to one of type T2. The Scratchpad II interpreter provides this information, for example, to the type resolve and modemap selection facilities. The facility is used when one needs to know absolutely when a coercion will be successful. An answer of "no," however, does not guarantee that a coercion could not be performed for specific data. For example, the system will respond "no" when asked whether an object of Polynomial(Integer) can be coerced to an object of Integer. As we saw above, though, retractions can be performed for constant polynomials.

The Resolve Facility

The resolve facility is used to determine a type T3 to which two types T1 and T2 can be coerced. It is used when an operation has homogeneous arguments (such as "+") or when a statement has several exit points and they must all return the same type (then and else clauses of an if statement, or multiple return statements in a function).

The resolve facility is symmetric: resolve(T1,T2) = resolve(T2,T1). The resolve facility is always successful because type Any is returned if a less general type cannot be found. Any is represented by a record with two components, the first being the original type of the object and the second being the object itself. Thus anything can be coerced to an object of type Any and resolve(Any,T) = Any for all T.

Two other types have special resolve rules. Type Void has but one object and is the type returned by such operations as variable declaration, function definition, if statements without else clauses and repeat loops. Several functions also return the object of type Void, including those that display things in two dimensional algebraic, TeX[3] and FORTRAN forms. Like Any, resolve(Void,T) = Void for all T. Type Exit is used for return statements and error statements. Its rule is resolve(Exit,T) = T for all T.

After some checks for special cases, the resolve facility has three components.

Resolve by Coercion Query

If T1 = T2, then the resolve of the pair is just T1. If T1 can be coerced to T2 and T2 cannot be coerced to T1, then resolve(T1,T2) = T2. If T1 and T2 are coerceable to one another, an arbitrary but canonical choice is made and returned.

Resolve by Rules

The interpreter has an internal database of rules which it tries to use to resolve two types. The rules are not complete, as they only attempt to take care of cases that cannot be dealt with in a more general way. Almost all the rules deal with polynomials. For example, one rule is

resolve(Polynomial(T1), UnivariatePoly(x,T2)) = resolve(Polynomial(T1),T2)

The variable of UnivariatePoly can be absorbed into the general constructor Polynomial and then the resolve facility is called again with different arguments. The second call may or may not use the rule system.

Resolve by Type Destructuring

This type of resolution is similar to the process involved in the coercion described in the last paragraph of "Coerce by Internal System Code". Given two towers of parametrized types, the interpreter tries to rearrange the towers and create a new type to which both of the original types are coerceable. This is a recursive process and involves using the coercion query facility to determine what tower rearrangements are possible.

An Example

As an example of coerce and resolve, we describe the inference involving in determining that the expression x + 1/2 evaluates to an object of the datatype Polynomial(RationalNumber). We assume x has not previously been given a value.

- Choose a default datatype of Symbol for x.
- Choose the datatype of Integer for 1 and 2.

[3] TeX is a trademark of the American Mathematical Society.

- Look for an operation "/" in Integer that has two arguments, each an integer. It is not found.
- Start a general search in the library for an operation "/" with two Integer arguments. One is found in RationalNumber and applied.
- Look for a "+" that takes a Symbol and a RationalNumber. None is found.
- Force x to an object of type Polynomial(Integer).
- Look for a "+" that takes a Polynomial(Integer) and a RationalNumber. None is found.
- Start a general search in the library for "+" operations. The only ones found take two arguments, each of the same datatype. Resolve Polynomial(Integer) and RationalNumber to get the datatype Polynomial(RationalNumber).
- Do the coercions
  Polynomial(Integer) → Polynomial(RationalNumber)
  RationalNumber → Polynomial(RationalNumber)
- Apply the "+" in the datatype Polynomial(RationalNumber) and return the result.

Acknowledgements

In addition to the authors, three people have contributed significantly to the development of the Scratchpad II interpreter. Scott C. Morrison (University of California, Berkeley) is responsible for the overall structure of the interpreter as it today, having largely rewritten this part of the system in 1984. Albrecht Fortenbacher (University of Karlsruhe) rewrote and greatly extended the resolve and coerce facilities in 1985. Michael Lucks (Southern Methodist University) contributed to the coerce and modemap selection facilities in 1986.

References

[1] Jenks, R. D. and Trager, B. M.. "A Language for Computational Algebra," *Proceedings of SYMSAC '81, 1981 Symposium on Symbolic and Algebraic Manipulation*, Snowbird, Utah, August, 1981. Also *SIGPLAN Notices*, New York: Association for Computing Machinery, November 1981, and *IBM Research Report* RC 8930 (Yorktown Heights, New York).

[2] Liskov, B., Atkinson, R., et al. *CLU Reference Manual*, New York: Springer-Verlag, 1981.

[3] Computer Algebra Group. *An Overview of the Scratchpad II Language and System*, Yorktown Heights, New York: IBM Corporation, April 1986.

[4] Sweedler, Moss E.. "Typing in Scratchpad II," *The Scratchpad II Newsletter*, Vol. 1, No. 2. Edited by R.S. Sutor, Yorktown Heights, New York: IBM Corporation, January 15, 1986.

[5] Jenks, R. D.. "A History of the SCRATCHPAD Project (1977-1986)," *The Scratchpad II Newsletter*, Vol. 1, No. 3, Edited by R.S. Sutor, Yorktown Heights, New York: IBM Corporation, May 15, 1986.

[6] Schaffert, C., Cooper, T., et al. "An Introduction to Trellis/Owl," *OOPSLA '86 Conference Proceedings, SIGPLAN Notices*, Volume 21, Number 11, New York: Association for Computing Machinery, November 1986, pp. 9-16.

[7] Jenks, R. D., Sutor, R. S., and Watt, S. M.. "Scratchpad II: An Abstract Datatype System for Mathematical Computation," *IBM Research Report* RC 12327 (Yorktown Heights, New York: November 17, 1986).

[8] Watt, S. M., and Jenks, R. D.. "Abstract Datatypes, Multiple Views and Multiple Inheritance in Scratchpad II," *The Scratchpad II Newsletter*, Vol. 1, No. 4, Edited by R.S. Sutor, Yorktown Heights, New York: IBM Corporation, March 15, 1987.