Completing the Compilation of SuchThat v0.7

H. Gast[1] S.Schupp R.Loos

{gasth,loos}@informatik.uni-tuebingen.de
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen, Germany

schupp@cs.rpi.edu
Department of Computer Science
Rensselaer Polytechnic Institute

[1]On leave of absence from the university of Tübingen

Contents

1 Introduction

1.1 Summary

SuchThat is a new programming language for generic programming. It combines the power of parameterization with the safety of type checking. In SuchThat it is possible to write

⟨decl.sth 4⟩ ≡
```
        Let R be a ring, R commutative;
        let P be the polynomials over R;
        p ∈ P; p is prime;◇
```
Macro never referenced.

and to treat the sentence *as a declaration*. As such declaration the sentence has to undergo the usual processing for declarations—starting from a scanning and parsing procedure and ending up with type checking and code generation. Programming in abstract concepts is perfectly possible.

SuchThat was proposed by the last two authors ([12]) who also agreed on implementing the language together with an entire system for generic programming. The SuchThat system under development includes a generic library, but also other components such as a verifier and an on-line documentation environment.

After a first prototype implementation it was decided in the late Fall 1996 to write a compiler from SuchThat to C^{++}, called st2cpp. The compiler construction was split into three parts, the syntactic analysis, the semantic analysis, and a back-end that translates to C^{++}. The language Scheme served as intermediate language. When the completion project, the project described here, was started, both the parser and the back-end were finished. For the semantic analysis—the major challenge for the SuchThat language—only single modules were available. There was a module to normalize so-called attributed structures, a module to process the complex SuchThat declarations, and a module to prove generic implications (in)correct. These modules were separately developed, partly by different people. It was unclear how smoothly they would fit together and whether major gaps in the specification of the compiler project or the language would appear. The task of the completion step was to integrate all existing modules into an executable compiler.

Achievements st2cpp was finished successfully in December 97. The experiments run with the translator allow to check and to refine decisions regarding the language design. It is now possible to give a formal definition of the language. Besides providing an executable program the major insight during the completion phase of st2cpp is the insight how *implications*, SuchThat's type checking paradigm, and instantiation have to work together.

Restrictions st2cpp does not translate the full SuchThat language. Language features that are currently either not supported or not yet publicly available are memory management, overload resolution, exception handling, and higher order functions. Furthermore, the SuchThat module language was entirely neglected.

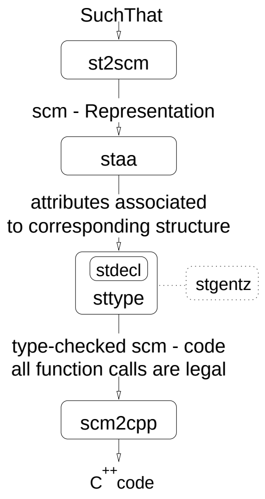
Next Steps The immediate next steps are to integrate the SuchThat garbage collection module, under active development, and to perform overload resolution for which the st2cpp program is fully prepared already. At the level of type checking st2cpp clarified how instantiation and implication have to be weaved. As a consequence, the implication module, `stgentz`, has to be modified correspondingly.

All parts of st2cpp, including the last completion steps, were developed in close cooperation of the two SuchThat groups at RPI and Tübingen, Germany. During the project described here the first author visited RPI. Parts of the project constitute his *Studienarbeit*, presented in Tübingen in December 1997.

1.2 An Overview over the Translator

The main goal of this project has been to integrate the existing modules of a translator from SuchThat to C^{++} and complete the implementation with the type-checking module.

Before we describe the latter part, we would like to give the reader an impression of the translation process as a whole and highlight the connecting points that had been specified beforehand.



SuchThat
st2scm
scm - Representation
staa
attributes associated to corresponding structure
stdecl
sttype
stgentz
type-checked scm - code all function calls are legal
scm2cpp
C++ code

Figure 1: Structure of the SuchThat to C^{++} translator

The translator itself is mostly written in Scheme[1] and most naturally also the intermediate representation of the SuchThat program have the form of Scheme lists. This makes reading and writing results particularly easy and encourages the division of the translator into separate stages.

1.3 The Frontend – st2scm

The language SuchThat has been designed to resemble the writing style found in mathematical textbooks. For this implementation, conventional tools, i.e. flex^{++} and bison^{++}, have been used to translate the SuchThat source into its intermediate representation. [9] contains the technical details.

As a broad summary we can say that the task of parsing a formalized, yet almost natural, language, yields the expected problems when specified as a LALR(1) grammar. The crucial points turned out to be the context-dependent meaning of tokens, which leads to the conclusion that LL parsing might be the more appropriate method, because of its inherent context-dependency of subexpressions.

1.4 Attribute Association – staa

One of the important features of the SuchThat structures sublanguage is the ability to give attributes to structures, which allows for a very detailed specification in particular of requirements of function parameters. An example is:

> Let T be a group, T is commutative.

Here *commutative* is an attribute of T.

Since these expressions are clearly context sensitive, they cannot be directly incorporated into a LALR(1) parser. staa[10] associates the attribute *commutative* with T and expresses their relation in the closed form

(with T commutative)

1.5 The Type Checker – sttype

A major part of this report is concerned with the type-checking process which guarantees type-safe calls to generic functions. Therefore at this point we only point out the questions answered in completing this project:

- To use STGENTZ in the process of type-checking it has to be extended to deal with bindings, i.e. extract the subexpressions which stand for type-variables in generic functions.
- An experimental implementation of this process, which is capable of demonstrating the requirements has been given in the function `instantiate`[1].
- The grounds for an implementation of overload resolution have been laid by finding all possible interpretations of an expression with respect to a given set of prototypes[2].
- Unlike e.g. in C++ template implementations, error and report messages give an intuitive and high-level understanding of type-errors; this demonstrates the usefulness of the both the approach of constrained generic functions and the particular implementation given here (see 2.2).

Since STTYPE also connects STAA to SCM2CPP, the intermediate code generated must match the input specifications of the backend. The code reflecting the necessary changes has been incorporated using STWEB macros whose names have the prefix `output adjust:`. Their definitions are given in section 9.

1.6 The BackEnd – scm2cpp

The target language of our translator is C^{++}, which makes the produced code executable on many systems in an efficient way. It is intended that the underlying library will use STL containers to implement the representations for the SuchThat structures; these containers and algorithms will also be available to the user.

The major challenges SCM2CPP[18] had to cope with are the flexible generation of type-expressions, in particular the generation of (template) function headers. Modifications to to the calling semantics (call-by-reference/value) as well as the descriptions of implemented C^{++} data-structures, including the needed header files and the number of template parameters, can be specified external to the code in configuration files.

2 Translation of Examples

In this section we demonstrate the features of the completed translator. The examples will show its ability to check constraints on calling generic functions effectively and ensure that minimal preconditions, which are stated as the type of parameters, are fulfilled.

The code in this section has been deliberately chosen from well-known focal points of generic programming to help the reader appreciate the decisions involved in accepting or rejecting a function call. We hope that the intuitive syntax of SuchThat will support understanding and make a detailed explanation of the notion of *instantiation* unnecessary at this very point (see section 8).

[1]The code of `instantiate` is not part of this report due to natural uncertainty about future modifications in the language design process.

[2]This process will be subject to changes and therefore the code will be contained in a separate report

2.1 Invocation of the Translator

The shell script ST2CPP as described in section A integrates the different modules of the translator. The usage is

```
st2cpp [-p] file
```

where *file* is the name *without* extension `.sth` of the SUCHTHAT source. To translate the example in file `gcd.sth` run

```
st2cpp gcd
```

The `-p` option stops the translation process after the parser stage which will be useful for incrementally building syntactically correct SUCHTHAT programs first and circumvent the time consuming type check during this phase.

2.2 Bindings and Substitution

The underlying concept of *instantiation* is the idea of pattern matching and substitution. However, in our case, the variables are constrained in what particular expressions they can be bound to and within these constraints, we can have further variables again. Why we need such a recursive understanding of substitutions is best explained by giving a comprehensive example. The writing style might remind the reader of Louis Reasoner [2] and the compiler corrects his mistakes.

Bindings Let us start with this problem: We want to write a function `BubbleSort` which can deal with finite sequences, sorting them in ascending order. Our first try looks like this:

- Declare *finite sequences* as a structure (=type).
- Point out that for sorting we need an *ordered* set.
- Introduce some local abbreviations.

"exinst.sth" 7a ≡

```
Global: Structure FiniteSequence over Set is abstract.
Algorithm: BubbleSort(a ; b)
        Let O be Set with O is ordered;
        Let FSQ(O) be FiniteSequence over O.
Input: a ∈ FSQ(O).
Output: b ∈ FSQ(O). ||

Algorithm: main()
Local: A ∈ Array(Integer).
(1) BubbleSort(A ; A) ||
◇
```

We get the message:

⟨example output 7b⟩ ≡

```
warning: in algorithm (main)
in expression (call1 bubblesort a out a)
in subexpression (bubblesort a)
(Implication not found
 ((app_par array integer) (over finitesequence (with set ordered)) . #t))◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

We want to assert that this implication holds and copy the given SCHEME list into `cache.imp.`, hoping that now the code compiles. It turns out though, that our function call is still not correct:

⟨example output 8a⟩ ≡

```
warning: in algorithm (main)
in expression (call1 bubblesort a out a)
(Implication not found
 ((over finitesequence (with set ordered)) (app_par array integer) . #t))
warning: in algorithm (main)
in expression (call1 bubblesort a out a)
(Output signature mismatch in
 (b (over finitesequence @o)) |= (a (app_par array integer)))
error: in algorithm (main)
in expression (call1 bubblesort a out a)
(No prototypes matching output signature
 (a)
 in pass 2)◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

What has happened ? *We have not been precise enough !* Our generic function is to return a structure that is *the very same finite sequence* that it received as an input structure. We therefore have to introduce a *binding* (**F**) for that sequence and rewrite the code as

`"exinst2.sth"` 8b ≡

```
Global: Structure FiniteSequence over Set is abstract.
Algorithm: BubbleSort(a ; b)
        Let O be Set with O is ordered;
        Let FSQ(O) be FiniteSequence over O;
        Let F be FSQ(O).
Input: a ∈ F.
Output: b ∈ F. ||

Algorithm: main()
Local: A ∈ Array(Integer).
(1) BubbleSort(A ; A) ||
◇
```

This compiles without errors. Looking at the reports generated, we find the messages:

⟨example output 8c⟩ ≡

```
*** report 2 #80 ***
(prototypes found for subexpressions are  #42)

*** report 2 #81 ***
((bubblesort a) :
 ((((bubblesort ((a @f)) out ((b @f)))
out
 (((b (app_par array integer))))
bindings
 ((((be f (over finitesequence @o)) (app_par array integer))
---
))))))◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

But where is the binding of `O`, our ordered set ? Let's examine the `.err` file again:

⟨example output 9a⟩ ≡

```
warning: in algorithm (main)
in expression (call1 bubblesort a out a)
in subexpression (bubblesort a)
(simplies-explicit? used structure-implication?
 (app_par array integer)  =>  (over finitesequence (with set ordered))
=> no bindings found)
◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

The `O` has not been bound, because the instantiation procedure was not able to match an `array` with a `finite sequence`. We could bind the *whole* structure to F (by the implication entered to the cache earlier), but we could not bind the *substructure* `O`.

This problem is a direct consequence of the separate treatment of implication and instantiation pointed out earlier. Therefore it will vanish as soon as the concepts are reunited.

Attributes The following example deals with lists and functions to manipulate them. It shows how attributes can be used to express preconditions.

We first introduce the notion of a homogeneous list and represent a (dense) univariate polynomial as a list of its coefficients. This is done to show how bindings carry over to the right hand side when referring to a structure declaration.

"exattr.sth" 9b ≡

```
Global: Structure List(Set) is implemented;
        Let R be Ring;
        Structure UnivariatePolynomial(R) is List(R).

Algorithm: f:=first(l)
        Let S be Set.
Input:  l ∈ List(S) with l is nonNull.
Output: f ∈ S. ||

Algorithm: r:=red(l)
        Let S be Set.
Input:  l ∈ List(S) with l is nonNull.
Output: r ∈ List(S). ||

Algorithm: c:=cons(f,r)
        Let S be Set.
Input:  f ∈ S;
        r ∈ List(S).
Output: c ∈ List(S) with c is nonNull. ||

Algorithm: main()
Local:  a ∈ UnivariatePolynomial(Integer);
        b ∈ List(Integer) with b is nonNull;
        c ∈ List(Integer);
        $x \in$ Float.
(1) a:=first(a).
(2) b:=red(b).
(3) c:=red(cons(3,b)).
(4) c:=cons(x,b) ||
◇
```

Obviously the statement (1) is wrong: a is a polynomial over the integers and thus a list of integers by declaration, but we miss the attribute `notNull`[3].

Here the way the `instantiation` puts it:

⟨example output 10a⟩ ≡

```
warning: in algorithm (main)
in expression (first a)
in subexpression (first a)
(Instanstantiation
 (app_par univariatepolynomial integer)  |=  (with (app_par list @s) nonnull)
failed because of missing attributes
 (nonnull))◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

The second expression is a legal function call, since b has the required attribute, however, the return value of `red` can be `null` even though the input was not. `red` destroys this property. Correspondingly we find the remarks:

⟨example output 10b⟩ ≡

```

warning: in algorithm (main)
in expression (red b)
(Instanstantiation
 (app_par list integer)  |=  (with (app_par list integer) nonnull)
failed because of missing attributes
 (nonnull))

error: in algorithm (main)
in expression (red b)
(No prototypes matching desired type
 ((with (app_par list integer) nonnull))
 in pass 2)

error: in algorithm (main)
in expression (red b)
(No prototypes with matching input signature)
◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

The statement in (3) is perfectly fine: `cons` returns non empty lists and `red` expects them. Here are the generated reports:

⟨example output 10c⟩ ≡

[3]However, such a property might follow from the context, for example from applying the Hoare calculus to the procedure. If we give the programmer the possibility to state such assertions, we can go ahead and include run-time requirements into the specification of the procedures. This would force the user to make explicit his assumptions and think about e.g. invariants.

```
*** report 2 #582 ***
(prototypes found for subexpressions are  #396)

*** report 2 #583 ***
((red (cons 3 b)) :
 ((((red ((l (with (app_par list @s) nonnull))) out ((r (app_par list @s))))
out
 (((r (app_par list integer))))
bindings
 ((((be s set) integer )
---
))))))

*** report 2 #584 ***
((cons 3 b) :
 ((((cons ((f @s) (r (app_par list @s))) out ((c (with (app_par list @s) nonnull
))))
out
 (((c (with (app_par list integer) nonnull))))
bindings
 ((((be s set) integer )
---
))))))

*** report 2 #585 ***
(3 :
 (((((3 integer) () out ())
out
 (((3 integer)))
bindings
 ())))))

*** report 2 #586 ***
(b :
 (((((b (with (app_par list integer) nonnull)) () out ())
out
 (((b (with (app_par list integer) nonnull))))
bindings
 ())))))
◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

The statement (4) is wrong for the reason that we try to insert a `Real` into a list of integers:

⟨example output 11⟩ ≡

```
warning: in algorithm (main)
in expression (cons x b)
in subexpression (cons x b)
(Inconsistent binding occured during merge of
 ((((be s set) real )
---
))
and
 ((((be s set) integer )
---
)))
error: in algorithm (main)
in expression (cons x b)
in subexpression (cons x b)
(No matching prototype found for expression
 (cons x b))
error: in algorithm (main)
in expression (cons x b)
(No prototypes with matching input signature)
◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

2.3 Interpreting Error Messages

The only two stages of compilation whose error messages are relevant to the user because they emerge from mistakes in the input, are the parser and the type-checker. The attribute association does not issue any messages and the back end, logically, gets only error free code.

The parser messages can be handled by scanning the line given with the error for syntactical mistakes. Keep in mind that sometimes an error emerges because of a forgotten delimiter in the previous declaration or statement.

STTYPE splits its messages into errors and warnings with the implicit connection that warnings without a following error can be ignored. Warnings indicate weak points which *might* lead to errors. In analyzing the source of an error the user can scan `sttype.err` backwards from the point of the error.

An explanation of the messages issued and the procedures they emerge from will be given together with a short description of the instantiation procedure in section 8. You can also find an piece of SuchThat code pointing out subtleties in the use of the language there.

3 Structure of the Module sttype

In the style of literate programming, we weave the source code and its documentation in the remaining sections. Also we will present the concepts in a top down approach, beginning with the basic input and output strategies of the whole module and defining the procedures used herein later on.

In the following sections we use the term "type" when referring to the general concept known from programming languages and the term "structure" when talking about the specific type system of SuchThat (see section 5.1).

3.1 Running sttype

We will describe the input STTYPE expects in terms of the high-level pattern language, which implements basically an LL(1) parser, taking advantage of the prefix notation of the intermediate language.

The "driver function" for using STTYPE as a filter becomes extremely simple: We first read the input via the standard input into a list and then analyze it using `interpret-pattern` with a description of the expected structure.

⟨the driver function 12⟩ ≡

```
(define the-file
  (do ((res '() res)
         (next (read) (read)))
        ((eof-object? next)
         (reverse res))
      (set! res (cons next res))))

(define result (interpret-pattern staa-out the-file #t))

(define output-port (open-output-file "sttype.out"))
⟨output adjust: write output as sequence 68d⟩
◇
```

Macro never referenced.

3.2 Reports and Errors

The issue of error and warning messages will be addressed by describing the environment in which the error occurred as precise as possible. Since line numbers etc. are lost in the intermediate representation, this is the only way to go in the current prototype implementation.

We will write reports and errors to these files:

⟨error and report handling 13a⟩ ≡
```
(define error-port  (open-output-file "sttype.err"))
(define report-port (open-output-file "sttype.log"))
◇
```
Macro defined by scraps 13ab, 14, 15abcd.
Macro never referenced.

These files will be written to using the functions `report`,`issue-error` and `issue-warning`. In order to allow the report output to be suppressed in a fine tuned way, we introduce *report levels* which generally correspond to certain functions.

Since the procedures involved into the type checking process are highly recursive, there must be a way to find corresponding procedure calls and coherent pieces of information. We introduce the *unique-report-number* for this purpose. It is printed in the form `#`*no* into the header of each report. The procedures keep a local copy of the number that corresponds to their first report and repeat it when issuing others. In this way the reports corresponding to one function call can be traced by using search facilities of a pager for example.

Algorithm:	report	
Input:	*level*	The numerical value describing the report level. `report-level-`*level* has to be `define`d globally.
	R	The report, which may be any Scheme object.
Side effects:		*R* together with a report header is written to the file specified by `report-port` if `report-level-`*level* is #t. The global variable `unique-report-nr` is incremented.

⟨error and report handling 13b⟩ ≡
```
(define report
  (lambda (level . R)
    (if (eval (string->symbol
                (string-append "report-level-" (number->string level))))
        (begin
          (display "\n*** report " report-port)
```

```
            (display level report-port)
            (display " #" report-port)
            (display unique-report-nr report-port)
            (display " ***\n" report-port)
            (display R report-port)
            (display "\n" report-port)
            (set! unique-report-nr (+ 1 unique-report-nr))))))
```
◇

Macro defined by scraps 13ab, 14, 15abcd.
Macro never referenced.

Errors and warnings concerning the compilation of the source code itself are written to the `error-port` using the following functions. As a general rule, errors show the consequence of previous warning messages. When receiving an error which is not immediately clear, the user should examine the warnings going backwards from the point of error.

Algorithm:	issue-error	
Input:	E	Any SCHEME object.
Side effects:		E together with an error header is written to `error-port`. The counter for errors is incremented.

Algorithm:	issue-warning	
Input:	W	Any SCHEME object.
Side effects:		W together with a warning header is written to the `error-port`. The counter for errors is incremented.

⟨error and report handling 14⟩ ≡
```
(define issue-warning
  (lambda W
    (display "warning: " error-port)

    (issue-where error-port)

    (display W error-port)
    (display "\n" error-port)
    (set! warning-count (+ 1 warning-count))))

(define issue-error
  (lambda E
    (display "error: " error-port)

    (issue-where error-port)

    (display E error-port)
    (display "\n" error-port)
    (set! error-count (+ 1 error-count))))
```
◇

Macro defined by scraps 13ab, 14, 15abcd.
Macro never referenced.

Algorithm:	issue-where
Side effects:	Write current location in the source file as close as possible as substitute for line numbers.

⟨error and report handling 15a⟩ ≡

```
(define issue-where
  (lambda (port)
    (if cur-algorithm
        (begin
          (display "in algorithm " port)
          (display cur-algorithm port)
          (display "\n" port)))
    (if cur-decl
        (begin
          (display "in declaration " port)
          (display cur-decl port)
          (display "\n" port)))
    (if cur-expr
        (begin
          (display "in expression " port)
          (display cur-expr port)
          (display "\n" port)))
    (if cur-subexpr
        (begin
          (display "in subexpression " port)
          (display cur-subexpr port)
          (display "\n" port)))))
◇
```

Macro defined by scraps 13ab, 14, 15abcd.
Macro never referenced.

Since line numbers of the original SuchThat program are not copied into the intermediate files, errors and warning messages will be described to the user in terms of the environment in the source file, that they appeared in. When entering a new declaration, a new algorithm or expression during the traversal of the input, the following variables will be set correspondingly:

⟨error and report handling 15b⟩ ≡

```
(define cur-decl #f)
(define cur-algorithm #f)
(define cur-expr #f)
(define cur-subexpr #f)
◇
```

Macro defined by scraps 13ab, 14, 15abcd.
Macro never referenced.

As mentioned earlier, we will count errors and warnings and keep track of the current unique report number.

⟨error and report handling 15c⟩ ≡

```
(define warning-count 0)
(define error-count 0)
(define unique-report-nr 1)
◇
```

Macro defined by scraps 13ab, 14, 15abcd.
Macro never referenced.

To suppress parts of the report messages, we define the following flags. A report message for level x will be issued only if `report-level-`x is #t.

⟨error and report handling 15d⟩ ≡

```
(define report-level-0 #t)
(define report-level-1 #t)
(define report-level-2 #f)
(define report-level-3 #t)
(define report-level-4 #f)
(define report-level-5 #f)
(define report-level-6 #t)
(define report-level-7 #f)
(define report-level-8 #f)
(define report-level-9 #f)
(define report-level-10 #f)
(define report-level-11 #t)
(define report-level-12 #f)
(define report-level-13 #f)

(define report-level-21 #f)
◇
```

Macro defined by scraps 13ab, 14, 15abcd.
Macro never referenced.

The following table describes the relation between functions and report levels. Note that some of the procedures are not described in detail in this document. Most important to the user of ST2CPP will be levels 0,1,2 and 12 to examine the source of errors more carefully.
If report level 2 is enabled, the produced messages will be copied into the output file with suffix `.imp`.

report level	output categories
0	debug messages / internal errors
1	found questions to the implication calculus
2	found function calls & matching prototypes
3	the recursive way through the file
4	processing of structure declarations
5	the `lookup-be-decl` function
6	the `instantiate` function
8	`apply-bindings-to-output`
7	the `fully-expand` function
9	expansion and lookup of aliases
10	matching a prototype's signature
11	trace of procedure `interpret-pattern`
12	trace of overload resolution
13	application of bindings
$2x$	examples

3.3 Reading the Input File

As mentioned earlier, the intermediate representation of the SuchThat program will be in a prefix format. This means that it can be easily "parsed" using LL(1) parsing techniques. We therefore describe a general and flexible approach to read the input file, in which the structure of a file can be directly described by patterns that are matched. Matching ends at procedures, which the user supplies and which are called with any object which is found in the position in the input file corresponding to the position of the procedure.

Since this is a prototype translator the specification of the input is likely to change, we have decided to implement a high level description of the input and match it against the actual file rather than provide a hard-coded procedural hierarchy.

3.3.1 The Pattern Language for Specifying the Input File

Since the input file is given in prefix notation, we can decide how to interpret it based on the first element of every single substructure. For example in the sequence

⟨example for prefix notation 17⟩ ≡

```
(if (< x y) (set! y 0))
(while (i<10) (set! i (+ i 1)))
◇
```

Macro never referenced.

we know that the first list is an `if` statement just by seeing the `if` and can distinguish its interpretation from the second one, since here we find `while` as the first element of the list.

So we can define the structure of the input file in the style of an LL-grammar, where every non-terminal (=list) is recognizable by its first element using the following language:[4]

pattern	$\rightarrow$	'(' pattern-procedure sub-patterns ')'
sub-patterns	$\rightarrow$	sub-patterns choice-list
		| sub-patterns handle-procedure
		| ϵ
choice-list	$\rightarrow$	'(' times default-proc choices ')'
choices	$\rightarrow$	choices '(' tag match-procedure sub-patterns ')
		| ϵ
times	$\rightarrow$	'iterate'
		| 'once'
		| 'optional'
tag	$\rightarrow$	SCHEME-STRING
default-proc	$\rightarrow$	SCHEME-PROCEDURE
		| ϵ
pattern-procedure	$\rightarrow$	SCHEME-PROCEDURE
handle-procedure	$\rightarrow$	SCHEME-PROCEDURE
match-procedure	$\rightarrow$	SCHEME-PROCEDURE

3.3.2 Semantics of a Pattern

A *pattern* (see above) is a grammar describing the expected structure of the input. Interpreting such a pattern means analyzing a given expression with respect to the productions of that grammar.

A short example for a pattern matching a SCHEME expression with "+" and "-" as operators is given below.

The `pattern-procedure` will be called whenever the reading of the `pattern` is *completed*, i.e. *after* all `sub-patterns` are recognized.

The `sub-patterns` describe a sequence of structures. Each one of them is either passed to a `handle-procedure` in the corresponding place in the pattern or matched with a choice out of a choice list.

`Choice-lists` correspond to the EBNF notations
`{ S1 | S2 | ... }*` for `'iterate`
`[ S1 | S2 | ... ]` for `'optional`
`{ S1 | S2 | ... }` for `'once`

where the choice, which of the `Si` will be used, is taken by the *tag*, i.e. the first element of the list. Given an list of *choices* $(S_1, ..., S_n)$ with *tags* $(t_1, ..., t_n)$ and a expression $(t ...)$ represented as a SCHEME list, the choice S_{i_0} is taken if string $\rightarrow$ symbol$(t_{i_0}) = t$.

[4] As usual, upper-case strings denote terminals, lower-case string non-terminals and ϵ is the empty string.

If no match is obtained this way, the *default-proc* is called with the expression. If *default-proc* is not specified, an error condition occurs for *once* patterns.

The matching of sub-structures in the input file against choices of a list will be interrupted when finding a non-list in the input or the input item does not match any of the choices given. In case of *once*, this is an error condition, for *iterate* we may want to achieve this.

An iteration is aborted if a non-list in the input is found as the next item. This is done to allow *iterate* descriptions with default-procs that still will terminate.

If there are no sub pattern entries in a *choice*, this *shortcut* will lead to calling the *match-procedure* with the entire original input. This may be used to simulate the procedure being called *before* the substructures are matched (see implementation in of *"define"* in the input description).

Example for iteration, the short-cut and choice lists:

⟨example for choice lists 18a⟩ ≡

```
Choice list:
(iterate
 ("A" proc1)
 ("B" proc2))
Input:
((A 1)
 (B 2)
 (A 3)
 (C 4))
◇
```

Macro never referenced.

In the given choice list, we do not find a *default-proc.* We find however, that one of the choices, "A", matches the first input non-terminal. The procedure *proc1* is called with input `(A 1)` (short-cut, the whole structure is passed). Since the choices are applied iteratively here, we can also match the second input and call *proc2* with `(B 2)` and again call *proc1* for the third item. However, the last element in the input list does not match any choices and no default-procedure is given, so the iteration stops, leaving `(C 4)` to be matched by the subpattern following our choice list in the pattern.

But how can recursive structures in the input file be matched ?

So far we are able to describe the file structure up to a finite level of depth. Truly recursive lists, i.e. patterns that are named and used within themselves, would lead to circular structures. The answer to our problem is *lazy evaluation*, meaning that references inside of a pattern to the same pattern are expanded only when needed. The `delay` and `force` concept (see [1, Sec.6.9] is not applicable however, since there is no predicate such as `delayed-object?` in the SCHEME standard.

Another SCHEME object, which delays the evaluation, is the `lambda` object, i.e. a procedure, whose body will not be evaluated until the function call and it happens to be the case that procedures are used to express recursion.

We therefore introduce a procedure `use-pattern` to be placed into a pattern as a *handle-procedure* and demonstrate its application in the following example, postponing the definition to section 4.

⟨example 18b⟩ ≡

```
(define ex-expression
  `(,unchanged ; don't treat the whole expression
    (once
     ,(lambda (x)
        (report 21 "Found leaf :" x)
        (list x))
        ("+" ,unchanged
         ,(use-pattern "ex-expression" #t)
         ,(use-pattern "ex-expression" #t))
        ("-" ,unchanged
         ,(use-pattern "ex-expression" #t)
         ,(use-pattern "ex-expression" #t)))))

(report 21 "example for use-pattern\n"
        ((use-pattern "ex-expression" #t) '(+ (- a b) c)))
◇
```

Macro never referenced.

This example produces the output:

⟨example-output 19a⟩ ≡

```
*** report 21 ***
(Found leaf : a)

*** report 21 ***
(Found leaf : b)

*** report 21 ***
(Found leaf : c)

*** report 21 ***
(example for use-pattern
 ((+ (- a b) c)))◇
```

Macro never referenced.

Any procedure supplied in patterns must conform to this specification:

Algorithm:		
Input:	x	x is the SCHEME object found in the input position corresponding to the position of the procedure.
Output:	y	y is a list (`list` z) where z is the output the procedure wants to be placed in the spot where the input x was found. y may be empty (see `cut-out`)
Side effects:		The procedure may update and change any global variables.

The return values of the procedures called in matching a sequence of sub-structures are `append`ed, which allows the user to cut out parts of the input that are not used in later stages of compilation.

Examples on how to use this feature are given below:

⟨predefined handler procedures 19b⟩ ≡

```
(define unchanged
  (lambda (x) (list x)))
(define cut-out
  (lambda (x) '()))
(define ignore
  (lambda (x) (list x)))
◇
```

Macro defined by scraps 19b, 34.
Macro never referenced.

3.4 Creating a Pattern Specification

Since it is assumed that modifications to the current state of the translator or a reuse of the pattern language will take place in the future, we include the following paragraphs, which are meant to provide the reader with hints and instructions on how to use the structure patterns as a useful tool. The reader who only wants to understand the present state may skip these remarks.

For describing a new input structure or to modify an existing pattern, we propose the following procedure:

1. Write down a LL(1) description in EBNF.
   This ensures that the method is appropriate and points out where special cases have to be treated.

   Pay attention to that a pattern can get ambiguous in the following case: For a pair of consequent choice lists which share a *tag* for some choice, the first list does not know about the second one.

2. Express the *top-level structure* as a pattern, using the *short-cut* for every sub pattern as *unchanged.* Also place *unchanged* in any spot that you would like to treat by a special procedure later.

   Make sure all the mandatory procedures are in place and none of the inserted procedures can be taken as an optional one. In this case set the optional procedure as *unchanged.*

3. Design the interface to read the file, call `interpret-pattern` and write the output.

4. Turn on report levels 11 and 0 and run the program on some sample input, that is surely correct. Internal errors that might occur indicate that something is wrong. You can get a hint by looking at the `.log` file: The last message is a `report 0`. Pay attention to the "(Choice ... returning ...)" messages, since they state, whether your sub patterns have been found and to the messages issued at the beginning of `interpret-`*x*.

5. Refine your description without semantic procedures such that your whole input is matched and returned. Think particularly about the wrap-feature, since mistakes are likely to happen.

6. Replace the *unchanged* procedures by the functions implementing the semantic check. Sharing data between two distinct function calls can be achieved by global variables, which might have to hold stacks, etc.

   Pay attention to the fact, that structure procedures (the ones after choices) are executed *after* parsing the substructures. For an example how to call a procedure *before* treating the substructures, look at the description of *"define"* which needs to save the header in the prototype table *"before"* scanning the body of the function.

7. Design the output as you want it, think especially of wrapping the results of your procedures into lists.

3.5 The Structure of the Input File

We are now going to describe the input to STTYPE using a pattern as described above. We hope that the structure of this meta language makes it easy to follow the implementation.

3.5.1 The File Level

On the top level of the input file we find `global` and `algorithm` definitions, which contain `variable`, `alias`, and `type` declarations.

We describe these in the following SCHEME list.

⟨define the structure of the input file 20⟩ ≡

```
(define stdecl-out
  `(,unchanged
    ,(use-pattern "file-structure" #f)
    ,unchanged ; global symbol-table)))

(define staa-out
  `(,unchanged
    ,(use-pattern "file-structure" #f)))

(define file-structure
  `(,unchanged
    (iterate ; choice list on file level: algorithms or global decl's
     ("global" ⟨output adjust: rename tag and delete empty "global" 68a⟩
      (iterate
       ("\\in" ,handle-global-var-decl
        ,unchanged ; left hand side
        ,unchanged ; right hand side
        )
       ⟨a global be declaration 40a⟩
       ⟨a global type declaration 45a⟩
       ))

     ("algorithm" ,handle-algorithm
      ,(lambda (x) ; read header to obtain algorithm name
         (set! cur-algorithm x)
         (list x))
      (iterate
       ⟨read local declarations 22a⟩
       ))

     ("define" ,(lambda (x)
                  (let ((ret x))
                    ⟨output adjust: set signature in ret 68f⟩
                    ⟨clear local tables for next algorithm 52a⟩
                    (list ret)))
      ,(lambda (x) ; read header to obtain algorithm name
         (set! cur-algorithm x)
         (set! define-ilst (cdr x))
         (set! define-name (car x))
         (list x))
      (iterate
       ⟨read local declarations 22a⟩
       )
       ; handle body: different from 'algorithm
      ,(lambda (x)
         ; all of the header has been read
         ; -> put to prototype table for recursive algorithms
         (handle-define-body x)
         (interpret-pattern
           `(,unchanged
             (once ⟨structure of the algorithm body 22b⟩))
           x #t))))))
◇
```

Macro never referenced.

Inside the algorithm header, we find `internal`,`input`, `output`, and `local` declarations. A pattern describing *one* such section is:

⟨read local declarations 22a⟩ ≡
```
("internal" ⟨output adjust: throw away internal declarations 65b⟩
  (iterate
   ,(lambda (x) (error "Not implemented\n" x "\nin internal section"))
   ⟨a local be declaration 39a⟩
   ⟨a local type declaration 43b⟩

   ))
 ("local" ⟨output adjust: rename tag to st-local 65d⟩
  (iterate
   ,unchanged ; default-proc
   ("\\in" ,(bind2nd handle-local-var-decl 'l)
    ,unchanged
    ,unchanged ; right hand side
    )))
 ("input" ⟨output adjust: rename tag to st-input 65e⟩
  (iterate
   ,unchanged ; default-proc
   ("\\in" ,(bind2nd handle-local-var-decl 'i)
    ,unchanged
    ,unchanged ; right hand side
    )))
 ("output" ⟨output adjust: rename tag to st-output 65f⟩
  (iterate
   ,unchanged ; default-proc
   ("\\in" ,(bind2nd handle-local-var-decl 'o)
    ,unchanged
    ,unchanged ; right hand side
    )))◇
```
Macro referenced in scrap 20.

Note that basically, the sections differ only in the `handle` procedure to be called.

3.5.2 The Algorithm Body

STAA passes the body of an algorithm definition unchanged and so the grammar for the input is derived from the description of the parser in the paper *A Translator* SUCHTHAT *to* SCHEME (see [9]). All we have to do is an almost literal translation of the BISON^{++} rules into our pattern language.

Due to a peculiarity in the parser a single return value is entered at the end of the algorithm body rather than in the `output` signature itself, which makes the code a legal SCHEME function call. We have to retrieve it now.

⟨structure of the algorithm body 22b⟩ ≡
```
("call-with-current-continuation" ,unchanged
 (once
  ("lambda" ,unchanged
  ,unchanged ; parameter list
  (once
   ("let" ,unchanged
    ,unchanged ; initialization of local variables
    ; Following : Statements of the body. They are all lists
```

```
        ⟨structure of a statement list 23a⟩
        (optional ; there might be a return value, which is the output-list
         ,(lambda (x) ; the output list
            (report 3 "Output list found:\n" x)
            (list x)
            ⟨output adjust: parser generated return value 68c⟩
            )))))))
◇
```

Macro referenced in scrap 20.

As a list of statements, corresponding to a *compound statement*, is a recursive structure in the way that it may contain *compound statements*, we define the following:

⟨structure of a statement list 23a⟩ ≡

```
(iterate
 ,(use-pattern "statement-structure" #t))◇
```

Macro referenced in scraps 22b, 23b, 24b.

A compound statement is either a single statement or a list of statements in a `(begin ...)` block.

⟨structure of a compound statement 23b⟩ ≡

```
(once
 ,(use-pattern "statement-structure" #t)
 ("begin" ,unchanged
  ⟨structure of a statement list 23a⟩
  ))◇
```

Macro referenced in scrap 24b.

In certain positions in statements we require return types from expressions, such as in `if` or `while` constructs we need a `boolean` return type.

⟨define the structure of a statement 23c⟩ ≡

```
(define boolean-expression
  (lambda (x)
    (next-expression-desired-types! '(bool) type-be-decls-l)
    (handle-expression x)))
;(trace boolean-expression)
◇
```

Macro defined by scraps 23cde, 24b.
Macro never referenced.

⟨define the structure of a statement 23d⟩ ≡

```
(define integer-expression
  (lambda (x)
    (next-expression-desired-types! '(integer) type-be-decls-l)
    (handle-expression x)))
;(trace integer-expression)
◇
```

Macro defined by scraps 23cde, 24b.
Macro never referenced.

⟨define the structure of a statement 23e⟩ ≡

```
(define check-variable-type
  (lambda (v var-env type type-env)
    (let ((lu (if (symbol? v) (find-var-lhs v var-env) #f)))
      (cond ((not lu)
             (issue-error "Declared variable of type "
                          (nice-type-structure type) " expected\n"
                          "found was " v))
            ((not (simplies-explicit? type type-env
                                      (get-var-decl-rhs lu)
                                      (get-var-decl-environment lu)))

             (issue-error "Variable " v " of wrong type\nexpected "
                          (nice-type-structure type)
                          "\nfound "
                          (nice-type-structure (get-var-decl-rhs lu))))))
    (list v)))
;(trace check-variable-type)
◇
```

Macro defined by scraps 23cde, 24b.
Macro never referenced.

Since we have to refer to a type found in one location, e.g. in a `case` statement, in other sub-structures, we use the global variables

⟨global variables 24a⟩ ≡

```
(define expression-desired-types #f)
(define expression-desired-type-e '())

(define next-expression-desired-types!
  (lambda (T env)
    (set! expression-desired-types T)
    (set! expression-desired-type-e env)))
◇
```

Macro defined by scraps 24a, 27d, 43a, 46b, 49, 50a, 51a.
Macro never referenced.

to communicate types and the corresponding environments (see section 8.2) across the borders of recursion. They are evaluated in the function `second-pass`.

Now that we have the special expression types defined, we may describe the structure of a statement very close to the input grammar as presented in [9].

⟨define the structure of a statement 24b⟩ ≡

```
(define statement-structure
  `(,unchanged
    (once
       ; Anything without a tag is a normal
       ; Scheme function call
     ,(lambda (x)
        (report 2 "Found statement-level function call " x)
        (handle-expression x))
     ("list"
      ; the parser generated return value
      ,(lambda (x)
         (let ((ret ⟨output adjust: parser generated return value 68c⟩))
            ret
            )))
      ("if" ,unchanged
       ,boolean-expression
       ⟨structure of a compound statement 23b⟩
       (optional
        ,(use-pattern "statement-structure" #t)
        ("begin" ,unchanged
         ⟨structure of a statement list 23a⟩
         )))
      ("display" ,unchanged)
      ("while" ,unchanged
       ,boolean-expression
       ⟨structure of a compound statement 23b⟩
       )
      ("for1" ,unchanged
        ;variable identifier (must be Integer)
       ,(lambda (x)
          (check-variable-type x (append symbol-table-l symbol-table-g)
                               'Integer type-be-decls-l))
       ,integer-expression ; first value (must be integer)
       ,integer-expression ; last value (must be integer)
       ⟨structure of a compound statement 23b⟩
       )
      ("for"  ,unchanged
        ;variable identifier (must be Integer)
       ,(lambda (x)
          (check-variable-type x (append symbol-table-l symbol-table-g)
                               'Integer type-be-decls-l))
       ,integer-expression ; first value (must be integer)
       ,integer-expression ; next value (must be integer)
       ,integer-expression ; last value (must be integer)
       ⟨structure of a compound statement 23b⟩
       )
      ("case" ,unchanged ; check that type of expression matches
       ,unchanged ; expression must be of type char / integer
       (iterate
        ; case clauses, each one is list of labels followed by (begin ...)
        (iterate
         ,(lambda (x) (report 3 "New case label" x) (list x))
         ("begin" ,unchanged
          ⟨structure of a statement list 23a⟩
          ))))
      ("set!" ,unchanged
       ; variable : its type determines the possible
       ;            return types of handle-expression in pass 2
       ,find-set!-lhs-type
       ,handle-expression ; the expression
       )
      ("return" ,unchanged
       ,unchanged ; return parameter == variable. Done be parser, not user.
       )
      ("exit" ,unchanged))))
```
◇

Macro defined by scraps 23cde, 24b.

Macro never referenced.

3.6 Special Functions for Particular Parts of the File

The structure of the file has now been described down to the parts that we must actually deal with in a flexible and extensible way, which facilitates future updates and changes.

3.6.1 Global Tables

All type and alias declarations will be held in global tables. Since in a declaration previously defined types and aliases, which are called the *environment* of the declaration, may be used, we have to extend the tables and still make visible to each declaration only those types and aliases, which have been defined before. In this process of course copying should be avoided, which leads to the following definition:

Algorithm:	extend-environment	
Input:	*environment*	a list of type and alias declarations.
	be-structure-decl	a single alias or type declaration constructed by the functions `make-be-decl` or `make-type-decl`.
Output:	*extended*	the environment *environment* augmented by the given declaration with • No copying has been done. • No existing references into *environment* can be used to access the new entry. • When searching through the whole output, the new entry will be found before any entry present in *environment*.

⟨the structure be-decl 26a⟩ ≡

```
(define extend-environment
  (lambda (environment be-structure-decl)
    (cons be-structure-decl environment)))
```
◇

Macro defined by scraps 26ab, 27c, 38.
Macro never referenced.

All alias and type declarations will be held in a common environment, which forces us to append a tag to each declaration stating its type. This tag may be retrieved using the following function:

Algorithm:	get-decl-tag	
Input:	*be-structure-decl*	An alias or type declaration constructed by the corresponding `make` functions.
Output:	*tag*	the tag of the declaration specifying its type, i.e. either `'be` or `'type`

⟨the structure be-decl 26b⟩ ≡

```
(define get-decl-tag
  (lambda (be-structure-decl)
    (car be-structure-decl)))
```
◇

Macro defined by scraps 26ab, 27c, 38.
Macro never referenced.

We also provide the following useful functions:

⟨the structure type-decl 27a⟩ ≡

```
(define type-be-decl?
  (lambda (x)
    (and (pair? x) (or (eq? (car x) 'be)
                       (eq? (car x) 'type)))))
(define type-decl?
  (lambda (x)
    (and (pair? x) (or (eq? (car x) 'type)))))

(define be-decl?
  (lambda (x)
    (and (pair? x) (or (eq? (car x) 'be)))))
◇
```

Macro defined by scraps 27ab, 42ab.
Macro never referenced.

Due to shadowing, the same list structure might still refer to different types or aliases. Therefore these are compared for equality using the functions:

Algorithm:	equal-structure-decls?	
Input:	*A*, *B*	are declarations with `type-decl? == #t`
Output:	*bool*	#t if *A* and *B* refer to the same declaration, #f otherwise.

⟨the structure type-decl 27b⟩ ≡

```
(define equal-structure-decls?
  (lambda (A B)
    (eqv? A B)))
◇
```

Macro defined by scraps 27ab, 42ab.
Macro never referenced.

Algorithm:	equal-be-decls?	
Input:	*A*, *B*	are declarations with `be-decl? == #t`
Output:	*bool*	#t if *A* and *B* refer to the same declaration, #f otherwise.

⟨the structure be-decl 27c⟩ ≡

```
(define equal-be-decls?
  (lambda (A B)
    (eqv? A B)))
◇
```

Macro defined by scraps 26ab, 27c, 38.
Macro never referenced.

Before actually reading in the declarations from the input, we need to create the global tables to hold the declarations:

⟨global variables 27d⟩ ≡

```
(define type-be-decls-g '())
(define type-be-decls-l '())
(define type-be-decls-d '())
◇
```

Macro defined by scraps 24a, 27d, 43a, 46b, 49, 50a, 51a.
Macro never referenced.

where the $-l$ environment is always an extension in the above sense of the $-g$ environment to handle shadowing in a convenient way and $-d$ is the set difference of $-l \setminus -g$.

4 Interpreting Structure Patterns

In section 3.3.2 we have described the semantics of a structured language to specify the input grammar in a very natural and easy-to-modify form. Now we are concerned with implementing an interpreter for this language.

4.1 The Top-level Procedure – `interpret-pattern`

The following procedure corresponds to the production

pattern $\rightarrow$ '(' pattern-procedure sub-patterns ')'

It handles in particular the pattern-procedure and calls *interpret-spatterns*.

Algorithm:	interpret-pattern	
Input:	*p*	A *pattern* as described in the grammar for patterns.
	x	The input to be parsed using the *p*.
	wrap?	If *x* is a single item, not a list of items to be matched against *p*, then it need to be wrapped in a list before matching to meet the specification of `interpret-spatterns`. This will be done if *wrap*=#t.
Output:	*out*	Depending on the user-supplied procedures as described in section 3.3.2. In general it is a concatenation of the returned values, which are inserted in the place corresponding to object, with which the user procedure had been called.
Side effects:		The pattern procedure gets called with the entire return value of the call to `interpret-spatterns`

⟨interpreter for the structure list 28⟩ ≡

```
(define ip-enter "interpret-pattern begin")
(define ip-leave "interpret-pattern end")

(define interpret-pattern
  (lambda (p x wrap?)
    (report 11 ip-enter "\nuse\n" (nice-pattern p) "\nfor\n" x)
    ; check for valid syntax in pattern
    (if (and (pair? p)
             (procedure? (car p)))
```

```
          (let ((ret (interpret-spatterns (if wrap? (list x) x) (cdr p))))
            ((car p) ret)
            (report 11 ip-leave x "\n->\n" ret)
            ret
               )
          (begin
            (report 0 "Ill formed pattern, expected "
                    "(procedure ...)\nfound\n"
                    p
                    )
            (error "Internal error")))))
◇
```

Macro defined by scraps 28, 29.
Macro never referenced.

4.2 The Procedure `interpret-spatterns`

`interpret-spatterns` handles the productions

sub-patterns	→	sub-patterns choice-list
		| sub-patterns handle-procedure
		| ϵ
choice-list	→	'(' times default-proc choices ')'
choices	→	choices '(' tag match-procedure sub-patterns ')
		| ϵ

The last one however is simply dispatched to the procedure `interpret-once-optional-iterate` and the second production is contained in the STWEB macro `interpret first subpattern`.

The description of the algorithm is contained in the semantics assigned to the productions: Items in the input are matched up against the grammar according to the definition of *sub patterns*.

Algorithm:	interpret-spatterns	
Input:	*expr*	The input to be matched by the *sub patterns*. It is a list of SCHEME symbols.
	spatterns	*sub patterns* as described in the grammar for patterns
Output:	*out*	concatenated return values of the user-specified functions or recursive calls to the interpreter procedures.

⟨interpreter for the structure list 29⟩ ≡

```
(define interpret-spatterns
  (lambda (expr spatterns)
    (report 11 "Interpret-spatterns\nexpr=\n"
            (if (null? expr)
                expr
                `(,(car expr) ...))
            "\nsubpatterns=\n"
            (nice-pattern
             (if (null? spatterns)
                 spatterns
                 `(,(car spatterns) ...))))
    (cond ((and (null? expr) (null? spatterns)) ; we're done
             '())
            ((and (null? spatterns )
                  (not (null? expr))) ; spatterns is to short
             (begin
               (report 0 "Syntax: spatterns too short\n"
                       spatterns
                       "\nExpression was left:\n"
                       expr
                       )
               (error "Internal error")))
            (; if expr null, check if no more procedure / once
             ; block occurs in spatterns which would be an error
             (and (null? expr)
                  (not (null? spatterns))
                  (do ((p spatterns (cdr p)))
                      ((or (null? p)
                           (procedure? (car p))
                           (and (pair? (car p))
                                (eq? (caar p) 'once)))
                       (not (null? p)))
                    ; empty body
                    ))
             (begin
               (report 0 "Syntax: too short expression\n" expr)
               (error "Internal error")))
            (#t ; match first sub-pattern and recurse
             ⟨interpret first subpattern 30⟩))))
     ◇
```

Macro defined by scraps 28, 29.
Macro never referenced.

The following macro decides, which production for *subpattern* to apply based on the first subpattern found. Its definition again is obvious from the description of the productions: We just have to distinguish the cases:

- procedure
- optional element
- match a single element
- iteratively match elements

⟨interpret first subpattern 30⟩ ≡

```
     (cond
      ; call to user-provided procedure
      ; call procedure with first item in expr and
      ; delete procedure & item in recursion
      ((procedure? (car spatterns))
       (report 11 "interpret-spatterns: calling procedure" (car spatterns)
               "\nfor expression\n" (car expr))

       ; append used to allow cutting out
       (append ((car spatterns) (car expr))
               (interpret-spatterns (cdr expr) (cdr spatterns))))
      ; (optional ...) sub structure : choose once if possible
```

```
    ; no break on (car e) not list
    ((and (pair? (car spatterns))
          (eq? (caar spatterns) 'optional))
     (interpret-once-optional-iterate
      'optional
      (get-default-proc-from-subpattern (car spatterns))
      (get-choices-from-subpattern (car spatterns))
      expr
      (cdr spatterns)))

    ; (once ...) same as (optional ...), match must be found though
    ((and (pair? (car spatterns))
          (eq? (caar spatterns) 'once))
     (interpret-once-optional-iterate
      'once
      (get-default-proc-from-subpattern (car spatterns))
      (get-choices-from-subpattern (car spatterns))
      expr
      (cdr spatterns)))

    ; iterate: recurse as long as possible over expr
    ;          break on non-list as subpattern & non-match
    ((and (pair? (car spatterns))
          (eq? (caar spatterns) 'iterate))
     (interpret-once-optional-iterate 'iterate
                                      (get-default-proc-from-subpattern (car spatterns))
                                      (get-choices-from-subpattern (car spatterns))
                                      expr
                                      (cdr spatterns)))
    (#t (report 0 "Found unexpected element in spatterns\n"
                (car spatterns))
        (error "Internal error")))
  ◇
```

Macro referenced in scrap 29.

4.3 The EBNF-like Productions

It seems obvious to handle all *choice* productions similarly, since they basically differ in the breaking and error conditions. So now we deal with the productions:

choices	→	choices '(' tag match-procedure sub-patterns ')
		| ϵ
times	→	'iterate'
		| 'once'
		| 'optional'
tag	→	SCHEME-STRING
default-proc	→	SCHEME-PROCEDURE
		| ϵ

The *shortcut* referred to later is the production where *sub-patterns* from the grammar above is empty, i.e. the entire substructure of the input is matched by calling the *match-procedure* which would normally be invoked with the return value of recursive matching.

which $\in$ {iterate,once,optional}, *default-proc* and *choices* as described earlier, *expr* the expression to be matched and *rest-pattern* the pattern which follows the *choice* and is to be used for later items in *expr* which are not contained in *choices*.

⟨interpret (car expr) which is optional / once / iterate block 32⟩ ≡

```
(define interpret-once-optional-iterate
  (lambda (which default-proc choices expr rest-pattern)
    (report 11 "interpret-once-optional-iterate"
            "\n\nwhich        = " which
            "\n\ndefault-proc = " default-proc
            "\n\nchoices      = " (let ((res '()))
                                   (do ((c choices (cdr c)))
                                       ((null? c) res)
                                     (set! res (cons (caar c) res))))

            "\n\nexpr         = " (if (null? expr)
                                     expr
                                     `(,(car expr) ...))
            "\n\nrest-pattern = " (nice-pattern rest-pattern))

    (cond ((and (null? expr)
                (or (eq? which 'iterate)
                    (eq? which 'optional)))
           '() ; we're done
           )
          ((null? expr) ; an error case, since 'once in subpattern
           (report 0 "Syntax of input: Expected\n" choices
                   "\nor call default procedure\n" default-proc)
           (error "Internal error"))
           (#t ; expr is not empty

        ; find the choice to be used
    (let* ((choice (if (and (pair? (car expr))
                            (symbol? (caar expr)))
                       (assoc (symbol->string (caar expr)) choices)
                       #f))
           (short-cut-used #f))
      (cond  ; "break" iterate on finding a non-list
             ((and (eq? which 'iterate)
                   (not (pair? (car expr))))
              (interpret-spatterns expr rest-pattern))

             ; interpret choice & default-proc
             (choice
              (report 11 "Made choice:\n" (nice-pattern (car choice))
                      "\nfor\n" (car expr))
              ; found entry => match sub-pattern
              ; and apply user-defined handler for pattern
              (if (not (and  (pair? (cdr choice))
                             (procedure? (cadr choice))))
                  (begin
                    (report 0 "Ill-formed subpattern\n" choice)
                    (error "Internal error")))

              (set! short-cut-used (null? (cddr choice)))

              (report 11 "Calling structure procedure " (cadr choice)
                 "\nwith\n" (car expr))
          (let ((r (append
                    ((cadr choice) ; handler for structure
                     (cons (caar expr)
                           (if short-cut-used
                               (cdar expr)
                               (interpret-spatterns (cdar expr)
                                                    (cddr choice)))))
          ; match next element in expression
          ; use same default-proc & choices if iterate
                    (if (eq? which 'iterate)
                        (interpret-once-optional-iterate
                         which default-proc choices (cdr expr) rest-pattern)
                        (interpret-spatterns (cdr expr)
                                             rest-pattern)))))
            (report 11 "Choice " (nice-pattern (car choice))
                    "\nreturning\n" r)
            r
               ))
         (default-proc ; default-proc if present
           (report 11 "Using default proc " default-proc
                   "\nfor\n" (car expr)
                   "\nchoices were\n" (nice-pattern choices))
           (append
            (default-proc (car expr))
            ; match next element in expression
            ; use same default-proc & choices if iterate
            (if (eq? which 'iterate)
                (interpret-once-optional-iterate
                 which default-proc choices (cdr expr) rest-pattern)
                (interpret-spatterns (cdr expr)
                                     rest-pattern))))
         (#t ; couldn't match, ok if optional / iterate
          (if (or (eq? which 'optional)
                  (eq? which 'iterate))
              (interpret-spatterns expr
                                   rest-pattern)
              (begin
                (report 0 "Syntax error in input:"
                       "Couldn't find match in\n"
                       choices
                       "\nfor\n"
                       (car expr))
                (error "Internal error"))))))))))
```
◇

Macro never referenced.

The following procedures are used just to clarify the access to the used parts of the pattern language here, because the access depends on the optional default procedure, which will be called whenever none of the choices match (DEFAULT-PROC $\rightarrow \epsilon$).

⟨choice list and default proc from subpattern 33⟩ ≡

```
(define get-choices-from-subpattern
  (lambda (spattern)
    (if (and (pair? (cdr spattern))
             (procedure? (cadr spattern)))
        (cddr spattern)
        (cdr spattern))))

(define get-default-proc-from-subpattern
  (lambda (spattern)
    (if (and (pair? (cdr spattern))
             (procedure? (cadr spattern)))
        (cadr spattern)
        #f)))
◇
```

Macro never referenced.

4.4 The Procedure `use-pattern`

To introduce recursive structures into our input, we may simply add a "virtual" non-terminal procedure that switches to another pattern. It may be used in place of a user-defined handler, since it returns a procedure with one argument, which takes the expression to be read. If wrap? is **#t** x will be put into a list, which allows processing of single structures with a choice-list.[5]

⟨predefined handler procedures 34⟩ ≡

```
(define up-begin "use-pattern: matching\n")
(define up-end   "use-pattern return:\n")
(define use-pattern
  (lambda (p1 wrap?)
    (if (not (string? p1))
        (begin
          (report 0 "use-pattern expected string")
          (error "Internal error")))
    (lambda (x)
      (report 11 up-begin p1)
      (let ((p ; and here's the delayed evaluation
             (eval (string->symbol p1))))
        (report 11 up-begin x "\nwith\n" (nice-pattern (cdr p)))
        (let ((ret (interpret-pattern p x wrap?)))
          (report 11 up-end ret)
             ret
             )))))
◇
```

Macro defined by scraps 19b, 34.
Macro never referenced.

5 Handling Declarations in the Input

In this part, we have to implement part of the functionality of STDECL. Declaration processing for a language as SUCHTHAT involves more decisions than are obvious for most programming languages.

We introduce the *technical* side primarily in this section, i.e. questions about which parts to expand when, and leave semantic definitions for the later sections.

[5]This is necessary whenever a single item of the input is to be matched to any item out of a choice list due to the specification of `interpret-spatterns` which requires its first argument to be a *list* of input items.

5.1 Allowed Input Structures for Types and Aliases

We give a description of the allowed input structures first to enable the reader to estimate the expressiveness of SuchThat. The structures recognized here are a subset of those accepted by the parser. However we have tried to pick an ample set of those and are confident, that other structures may be easily added.

```
      structure  →  IDENTIFIER
                    | (with structure IDENTIFIER-LIST)
                    | (of first-structure structure)
                    | (over first-structure structure)
                    | (app_square first-structure structure-list)
                    | (first-structure structure-list)
                    | (^ first-structure structure)
                    | (_ first-structure structure)
 structure-list  →  structure-list structure
                    | structure
first-structure  →  IDENTIFIER
                    | structure
```

If

$$\textit{first-structure} \rightarrow \text{IDENTIFIER}$$

then in the `instantiate` and `lookup-`*xxx* functions these identifiers of two structures being matched have to be `eq?` and will *not* be looked up in the environment.

This prevents the user from "renaming" built-in data types without referring to their entire definition in a proper alias-declaration.

Examples for structures could be

```
   (with ring commutative zero_divisor_free)
(over array (with set ordered))
(app_square map (with S ordered) T)
```

The last one is the translation of

```
Map[S,T] with S is ordered
```

where `S` and `T` are declared earlier to be e.g. sets.

Definition 1 *For the left hand side of a* be *declaration we impose the following constraints:*

- *Parameters of structures may not be parameterized structures.*
- *No abstract types*[6] *may be used in parameters.*
- *Any bindable identifier*[7] *may be only used* once.

Obviously the first two conditions are equivalent as to say:
The production first-structure→ *IDENTIFIER may be applied only once in a left hand side of a* be *declaration and*
every identifier that is not a first-structure, *must be a* bindable identifier.

Intuitively, this gives us all linear, i.e. non-nested structures on the SuchThat level, which seems appropriate to enforce the use the *be* declaration for aliases and macros. We incorporate this into the following predicate, simple top level identifiers being bindable identifiers which have to be accepted.

[6] Identifiers which have not been introduced by a declaration.
[7] An identifier introduced in a `Let` declaration

⟨check constraint upon left hand side of a *be* declaration 36⟩ ≡

```
(define check-be-lhs
  (lambda (lhs env)
    (cond ((symbol? lhs) #t)
          (#t ; a structure, def. applies.
           (and (check-be-lhs1 lhs env)
                (check-be-lhs2 lhs))))))

; return symbol if first structure is found,
; #t if ok structure, #f if constraint violated
(define check-be-lhs1
  (lambda (s env)
    (cond ((symbol? s) ; must be bindable
           (not (not (lookup-alias s env))))
          ((pair? s)
           (let ((first-structure (if (symbol? (cadr s)) (cadr s) #f))
                 (res #t))
             (for-each
              (lambda (sub)
                (let ((r (check-be-lhs1 sub env)))
                  (if (symbol? r)
                      (if first-structure
                          (set! res #f)
                          (set! first-structure r))
                      (set! res (and res r)))))
              (if (symbol? (cadr s)) (cddr s) (cdr s)))
                (if (and res first-structure)
                    first-structure
                    res)))
          (#t #f))))

; check if identifiers are not used twice
; pre: all non-first-structure identifiers are bindable
; ret: list of ids or #f
(define check-be-lhs2
  (lambda (lhs)
    (cond ((symbol? lhs) (list lhs))
          (#t (let ((res '()))
                (for-each
                 (lambda (sub)
                   (set! res
                         (union-of-disjunctive-id-sets
                          res
                          (check-be-lhs2 sub))))
                 (if (symbol? (cadr lhs))
                     (cddr lhs)
                     (cdr lhs))))))))

(define union-of-disjunctive-id-sets
  (lambda (A B)
    (cond ((or (not A) (not B)) #f)
          ((null? A) B)
          ((memq (car A) B) #f)
          (#t (let ((rec (union-of-disjunctive-id-sets (cdr A) B)))
                (if rec
                         (cons (car A) rec)
                         #f))))))
◇
```
Macro never referenced.

In several places, we will have to separate attributes introduced by the `with` tag and deal with them separately. The following procedures generalize the approach:

⟨strip off the with tags from structures 37a⟩ ≡
```
(define attributed?
  (lambda (s)
    (and (pair? s) (eq? (car s) 'with))))

(define strip-with
  (lambda (s)
    (if (attributed? s)
        (cadr s)
        s
        )))

(define get-with-attributes
  (lambda (s)
    (if (attributed? s)
        (cddr s)
        '())))
◇
```
Macro never referenced.

Definition 2 *In declarations introducing variables, bindable identifiers and structures without parameters possible attributes are transferred from the left hand side to the right hand side.*

This enables us to attach attributes to the whole structure on the right hand side of a *be* declaration, which would otherwise be syntactically impossible on the SuchThat level.

5.1.1 Preprocessing the Structures

Note that all of the allowed structures carry a unique (car) field if they are not identifiers, except for

(*first-structure structure-list*)

which is the translation for

first-structure **(***structure-list***)**

Thus we translate an input expression with the following function, which analogously to the tag `app_square` appends the tag `app_par`(entheses) to these structures.

⟨preprocessor for structures to force unique tags 37b⟩ ≡
```
(define known-structure-tags '(with of over app_square app_par ^ _))

(define preprocess-structure
  (lambda (e)
    (cond ((not (pair? e)) e)
          ((member (car e) known-structure-tags)
           (do ((res (list (car e)) res)
                (l (cdr e) (cdr l)))
               ((null? l) (reverse res))
             (set! res (cons (preprocess-structure (car l))
                             res))))
          (#t (do ((res (list 'app_par) res)
                   (l e (cdr l)))
                  ((null? l) (reverse res))
                (set! res (cons (preprocess-structure (car l))
                                res)))))))
◇
```

Macro never referenced.

5.2 Alias Declarations

Although the exact semantics of an *alias* in SuchThat will be discussed later in section 6 we can already read in declarations at this point assuming an intuitive idea about the meaning of `expand-aliases`

Aliases are short-cuts for longer structure declarations. In this way they are also appropriate to deal with *partial instantiation* .

In expanding all aliases we obtain a structure.

We decide to rather group together all procedures for reading in the file than to scatter them about the whole of the text. However, the reader is encouraged to read about the semantics of aliases before continuing in this section.

We define the *syntactic* structure of a *be*-declaration:

1. the *left hand side*, which is a structured type.
2. the *right hand side*, which is a structured type and has been expanded using earlier *be*-declarations.
3. the types and be declarations defined earlier in the file, as they provide the basis for expanding aliases and the constraints upon binding. These are also called the *environment*.

We abbreviate a declaration

Let A be B.

by

$$A \leftrightarrow B$$

Be-declarations can be used transitively as the right hand side will be expanded on reading.

As we will hold types and be-declarations in one list, the current environment, we have to attach a tag to distinguish them.

We require, that both the left *and* right hand side of a *be* declaration be expanded. This way aliases are applied transitively.

⟨the structure be-decl 38⟩ ≡

```
(define make-be-decl
  (lambda (lhs rhs environment)
    (list 'be lhs rhs environment)))
```

```
(define get-be-decl-lhs
  (lambda (be-decl)
    (cadr be-decl)))

(define get-be-decl-rhs
  (lambda (be-decl)
    (caddr be-decl)))

(define get-be-decl-environment
  (lambda (be-decl)
    (cadddr be-decl)))
◇
```

Macro defined by scraps 26ab, 27c, 38.
Macro never referenced.

We read in the actual declaration from the input with the following part of a pattern:

⟨a local be declaration 39a⟩ ≡

```
("be" ,handle-local-be-decl)◇
```

Macro referenced in scrap 22a.

⟨reading a local be-declaration 39b⟩ ≡

```
(define handle-local-be-decl
  (lambda (x)
    (let ((urn (string-append "#" (number->string unique-report-nr))))
    (report 4 "reading local be declaration " urn "\n" x)
    (set! cur-decl x)

    (let ((lhs (preprocess-structure (cadr x)))
          (rhs (expand-aliases (preprocess-structure (caddr x))
                               type-be-decls-l))
          (rhs-known? #f))
      ⟨transfer 'with from lhs to rhs 44a⟩
      ⟨lookup rhs for be declaration 41a⟩
      (cond ⟨check legal be declaration 41b⟩
            (#t
             (if (not (symbol? lhs))
                 (set! lhs (expand-bindable-ids lhs type-be-decls-l)))

             (if (bindable-identifier? lhs) ; return to old state
                 (set! lhs (get-be-decl-lhs lhs)))

             (report 4 urn " expanded to\n"
                     (nice-type-structure lhs) " BE "
                     (nice-type-structure rhs))
             (set! type-be-decls-l
                   (extend-environment type-be-decls-l
                                       (make-be-decl lhs rhs
                                                     type-be-decls-l)))
             (set! type-be-decls-d
                   (extend-environment type-be-decls-d
                                       (make-be-decl lhs rhs
                                                     type-be-decls-l)))
```

```
                (set! cur-decl #f)

                ; (list x) commented because of output adjust
                ))))  ; let ((urn))
        ⟨output adjust: throw away "be" declarations 67c⟩
        ))
    ◇
```

Macro never referenced.

⟨a global be declaration 40a⟩ ≡

```
    ("be" ,handle-global-be-decl)
    ◇
```

Macro referenced in scrap 20.

⟨reading a global be-declaration 40b⟩ ≡

```
    (define handle-global-be-decl
      (lambda (x)
        (set! cur-decl x)
        (let ((urn (string-append "#" (number->string unique-report-nr))))
        (report 4 "reading global be declaration " urn "\n" x)
        (let ((lhs (preprocess-structure (cadr x)))
              (rhs (expand-aliases (preprocess-structure (caddr x))
                                   type-be-decls-g))
              (rhs-known? #f))
          ⟨transfer 'with from lhs to rhs 44a⟩
          ⟨lookup rhs for be declaration 41a⟩
          (cond ⟨check legal be declaration 41b⟩
                (#t
                 (if (not (symbol? lhs))
                     (set! lhs (expand-bindable-ids lhs type-be-decls-l)))

                 (report 4 urn " expanded to\n"
                         (nice-type-structure lhs) " BE "
                         (nice-type-structure rhs))
                 (set! type-be-decls-g
                       (extend-environment type-be-decls-g
                                           (make-be-decl lhs rhs type-be-decls-g)))
                 (set! type-be-decls-l type-be-decls-g)
                 (set! type-be-decls-d '())
                 (set! cur-decl #f)

                 ;(list x))))) ; (let ((urn)))
        ⟨output adjust: throw away "be" declarations 67c⟩
        ))
    ◇
```

Macro never referenced.

The expanding of bindable identifiers is indeed a special case of the regular expand procedure and we could introduce a parameter to `expand-aliases` restricting its work to identifiers. However, for a prototype implementation it seems more appropriate to keep procedures separate.

⟨expanding bindable identifiers 40c⟩ ≡

```
(define expand-bindable-ids
  (lambda (s env)
    (cond ((symbol? s)
           (let ((lu (lookup-alias s env)))
             (if lu
                 (car lu)
                 s)))
          (#t (if (symbol? (cadr s))
                  (append (list (car s) (cadr s))
                          (map (bind2nd expand-bindable-ids env)
                               (cddr s)))
                  (cons (car s)
                        (map (bind2nd expand-bindable-ids env)
                             (cdr s))))))))
◇
```

Macro never referenced.

⟨lookup rhs for be declaration 41a⟩ ≡

```
(set! substructure-undefined-user-error #t)
(set! rhs-known? (not (not (lookup-type rhs type-be-decls-l))))
(set! substructure-undefined-user-error #f)
◇
```

Macro referenced in scraps 39b, 40b.

⟨check legal be declaration 41b⟩ ≡

```
((and (not rhs-known?) (not (abstract-type? rhs)))
 (issue-error "type of right hand side unknown")
 '())

((not (check-be-lhs lhs type-be-decls-l))
 (issue-error "Constraints upon alias lhs violated")
 (set! cur-decl #f)
 '())
◇
```

Macro referenced in scraps 39b, 40b.

5.3 Structure Declarations

Again we have to postpone the exact meaning of a *structure* in SuchThat to be able to present the procedures in a compact way. The reader is referred to section 8.2 for a discussion of the concepts.

The `lookup-type` function, which is used to determine the *meaning* of the right hand side of declarations roughly speaking tries to find a type declaration in an environment such that the given structure is a particular instance of the left hand side of that declaration.

Again we first describe the syntactical structure of a type declaration. A type declaration consists of the following parts:

- The *left hand side*, i.e. the structure to be declared.
- The *right hand side*, i.e. a already known structure *used* to define the *left hand side*.
- The declaration, that the right hand side refers to (its *lookup*) or #f if the right hand side is an abstract type, i.e. an identifier not previously declared.

- The *environment* in which the declaration takes place. This is simply a list of other *structure* declarations which determine the set of known structures.

For a structure declaration

> **Structure** A **is** B.

we write

$$A \to B$$

We describe the syntactical structure by introducing `make-structure-decl`:

⟨the structure type-decl 42a⟩ ≡

```
(define make-structure-decl
  (lambda (lhs rhs lookup environment)
    (list 'type lhs rhs environment lookup)))
◇
```

Macro defined by scraps 27ab, 42ab.
Macro never referenced.

Note that *lhs* and *rhs* have a yet undefined structure. They are supposed to be *user structures*. In case of the right hand side this also includes built-in types of course.

⟨the structure type-decl 42b⟩ ≡

```
(define get-structure-decl-lhs
  (lambda (type-decl)
    (cadr type-decl)))

(define get-structure-decl-rhs
  (lambda (type-decl)
    (caddr type-decl)))

(define get-structure-decl-environment
  (lambda (type-decl)
    (car (cdddr type-decl))))

(define get-structure-decl-lookup
  (lambda (type-decl)
    (car (cddddr type-decl))))
◇
```

Macro defined by scraps 27ab, 42ab.
Macro never referenced.

Reading a type declaration coarsely consists of the following steps:

1. Expand the left and right hand side using aliases.
2. Check for re-declaration.
3. Check that the right hand side is known or an abstract type.
4. Enter the declaration in the corresponding table(s).

The following variable is necessary, since we normally consider it an invariant that all substructures of a type are known and it is thus an internal error due to the violation of this invariant if they are not. When reading declarations however, the user is responsible for referring to defined types only.

⟨global variables 43a⟩ ≡

```
(define substructure-undefined-user-error #f)
◇
```

Macro defined by scraps 24a, 27d, 43a, 46b, 49, 50a, 51a.
Macro never referenced.

The overall structure looks as follows:

⟨a local type declaration 43b⟩ ≡

```
("type" ,(lambda (x)
         (let ((env type-be-decls-l)
               (lhs (cadr x))
               (rhs (caddr x))
               (urn (string-append "#" (number->string unique-report-nr))))

           (report 4 "reading local type-decl\n" x)
              ⟨preprocess and expand declaration 43c⟩
              (set! substructure-undefined-user-error #t)

              (let* (
                     ⟨lookup type declarations of left and right hand side 44b⟩
                        )
                (set! substructure-undefined-user-error #f)
                (cond ⟨check type redeclaration 44c⟩
                      ⟨enter local type declaration to environment 44e⟩
                      ⟨error : right hand side unknown 44d⟩
                 )))
         ⟨output adjust: throw away "type" declarations 68b⟩
         ))◇
```

Macro referenced in scrap 22a.

The steps laid out above are implemented in the following macros. Aliases are expanded when finding them, so before entering the declaration into our table we modify it this way:

⟨preprocess and expand declaration 43c⟩ ≡

```
(set! cur-decl x)

; preprocess structures to a uniform tagging
(set! lhs (expand-aliases (preprocess-structure lhs)
                          type-be-decls-l))
(set! rhs (expand-aliases (preprocess-structure rhs)
                          type-be-decls-l))

; any with attributes of the left hand side go to the right hand side
; type declarations are NOT structure implications
⟨transfer 'with from lhs to rhs 44a⟩
(report 4 urn "after expanding " (list (nice-type-structure lhs)
                                       (nice-type-structure rhs)))
◇
```

Macro referenced in scraps 43b, 45a.

Due the definition of attributes on page 35 we define the following:

⟨transfer 'with from lhs to rhs 44a⟩ ≡
```
(if (and (attributed? lhs)
         (symbol? (strip-with lhs)))
    (let ((attr (filter-list symbol?
                             (append (get-with-attributes lhs)
                                     (get-with-attributes rhs)))))
      (set! lhs (strip-with lhs))
      (if (not (null? attr))
          (set! rhs (append (list 'with (strip-with rhs)) attr)))))
◇
```
Macro referenced in scraps 39b, 40b, 43c, 48.

⟨lookup type declarations of left and right hand side 44b⟩ ≡
```
(lu1 (lookup-type rhs env))
(decl (if lu1 (car lu1) #f))
(lu2 (lookup-type lhs env))
(redecl?  (if lu2 (car lu2) #f))
◇
```
Macro referenced in scraps 43b, 45a.

⟨check type redeclaration 44c⟩ ≡
```
(redecl? (issue-error "Redeclaration in\n" x)
                      (set! cur-decl #f)
                      '())
◇
```
Macro referenced in scraps 43b, 45a.

⟨error : right hand side unknown 44d⟩ ≡
```
(#t (issue-error "Unknown type used in declaration\n"
                 (list (nice-type-structure lhs)
                       (nice-type-structure rhs)))
    (report 4 "Unknown type used in declaration\n"
            (list (nice-type-structure lhs)
                  (nice-type-structure rhs)))

    (set! cur-decl #f)
    '()) ; cut it out => lhs is undeclared
◇
```
Macro referenced in scraps 43b, 45a.

⟨enter local type declaration to environment 44e⟩ ≡
```
; strip with before comparing:
; attributes are always allowed.
((or (abstract-type? (strip-with rhs)) decl)
; enter the decl into the local environment
 (report 4 urn "entering to local environment")

 (set! type-be-decls-l
       (extend-environment type-be-decls-l
                               (make-structure-decl lhs
                                                    rhs
                                                    decl
                                                    type-be-decls-l)))
       (set! type-be-decls-d
             (extend-environment type-be-decls-d
                               (make-structure-decl lhs
                                                    rhs
                                                    decl
                                                    type-be-decls-l)))
       (report 4 urn "environment is now "
               (nice-environment type-be-decls-l))
       (set! cur-decl #f)
       (list x))
      ◇
```

Macro referenced in scrap 43b.

In the global case we almost do the same, except that entering the new declaration into the tables is different. We therefore may reuse the macros defined earlier.

⟨a global type declaration 45a⟩ ≡

```
      ("type" ,(lambda (x)
                 (let ((env type-be-decls-g)
                       (lhs (cadr x))
                       (rhs (caddr x))
                       (urn (string-append "#" (number->string unique-report-nr))))

                   (report 4 "reading global type-decl\n" x)
                   ⟨preprocess and expand declaration 43c⟩
                    (set! substructure-undefined-user-error #t)

                    (let* (
                           ⟨lookup type declarations of left and right hand side 44b⟩
                              )
                      (set! substructure-undefined-user-error #f)
                      (cond ⟨check type redeclaration 44c⟩
                            ⟨enter global type declaration to environment 45b⟩
                            ⟨error : right hand side unknown 44d⟩
                            )))
                 ⟨output adjust: throw away "type" declarations 68b⟩
                 ))◇
```

Macro referenced in scrap 20.

Extending the global environment and making the local an extension of the global environment are carried out in the next piece of code:

⟨enter global type declaration to environment 45b⟩ ≡

```
      ((or (abstract-type? (strip-with rhs)) decl)
      ; enter the decl into the global environment
       (report 4 urn "entering to global environment")
       (set! type-be-decls-g
             (extend-environment type-be-decls-g
                               (make-structure-decl lhs
                                          rhs
                                          decl
                                          type-be-decls-g)))
  (report 4 urn "environment is now \n"
          (nice-environment type-be-decls-g))

  ; make local environment extension
  (set! type-be-decls-l type-be-decls-g)
  (set! type-be-decls-d '())

  (set! cur-decl #f)
  (list x))
◇
```

Macro referenced in scrap 45a.

5.4 Variable and Constant Declarations

Variables are represented by identifiers and do have a type, that describes the value they may hold more precisely than in conventional programming languages.

In SuchThat numbers can be constants and have to be declared for a precise definition of the semantics of expressions.[8] We define therefore:

⟨definition of a SuchThat constant 46a⟩ ≡

```
(define is-st-constant? number?)
◇
```

Macro never referenced.

We use global tables to hold `global` and `local` declarations since SuchThat has got only 2 scopes. We need to partition the local declarations in a way that we may distinguish input and output parameters. The `-i` and `-o` lists will contains the input and output parameters. They are subsets of the `-l` table which will be used whenever the section, in which a variable is declared, does not matter.

⟨global variables 46b⟩ ≡

```
(define symbol-table-g '())
(define symbol-table-l '())
(define symbol-table-i '()) ; for input
(define symbol-table-o '()) ; for output
◇
```

Macro defined by scraps 24a, 27d, 43a, 46b, 49, 50a, 51a.
Macro never referenced.

The right hand sides of variable declarations will be expanded using aliases as any declaration is expanded upon reading.

As an invariant, the lists whose names end with `-i` and `-o` are subsets of `-l`. The `-l` list contains local declarations together with links to the input and output parameters.

They hold lists of the structure `var-decl` which is defined by the following functions:

⟨the structure of a variable declaration 46c⟩ ≡

```
(define make-var-decl
  (lambda (lhs rhs type-environment)
```

[8]Integer constants can be used without declaration, since they are common as indexes for loops, etc.

```
        (list lhs rhs type-environment)))

    (define get-var-decl-lhs
      (lambda (vd)
        (car vd)))

    (define get-var-decl-rhs
      (lambda (vd)
        (cadr vd)))

    (define get-var-decl-environment
      (lambda (vd)
        (caddr vd)))
    ◇
```

Macro defined by scraps 46c, 47.
Macro never referenced.

The following access functions will be used when searching a table for a variable / constant declaration.

⟨the structure of a variable declaration 47⟩ ≡

```
    (define find-var-lhs
      (lambda (lhs decls)
        (cond ((null? decls) #f)
              ((eq? lhs (get-var-decl-lhs (car decls)))
               (car decls))
              (#t (find-var-lhs lhs (cdr decls))))))

    (define find-var-rhs
      (lambda (rhs decls)
        (cond ((null? decls) #f)
              ((eq? rhs (get-var-decl-rhs (car decls)))
               (car decls))
              (#t (find-var-rhs rhs (cdr decls))))))

    ; find all definitions
    ; return #f if none is found, else the list
    (define find-var-lhs-all
      (lambda (lhs decls)
        (cond ((null? decls) #f)
              ((eq? lhs (get-var-decl-lhs (car decls)))
               (let ((r (find-var-lhs-all lhs (cdr decls))))
                 (if r
                     (cons (car decls) r)
                     (list (car decls)))))
              (#t (find-var-lhs-all lhs (cdr decls))))))
    ◇
```

Macro defined by scraps 46c, 47.
Macro never referenced.

In reading a variable or constant definition we perform the four steps:

1. If we deal with a variable, we check for redeclaration.[9]

[9]Constants can be overloaded, so they might be declared more than once.

2. We expand the aliases on the right hand side to obtain the type.
3. If the right hand side is a parameterized structure, it must have been declared previously.
4. We enter the declaration into either the local or global environment.

⟨functions for special file parts 48⟩ ≡
```
(define handle-global-var-decl
  (lambda (x)
    (set! cur-decl x)
    (let ((urn (string-append "#" (number->string unique-report-nr)))
          (res '()))
    (report 3 "reading global variable decl: " urn "\n" x)

    ; (1)
    (if (and (find-var-lhs (cadr x) symbol-table-g)
             (not (is-st-constant? (cadr x))))
        (issue-error "Redeclaration of variable " (cadr x))
         ; (2)
        (let ((lhs (cadr x))
              (rhs (expand-aliases (preprocess-structure (caddr x))
                                    type-be-decls-g))
              (env type-be-decls-g))
          ⟨transfer 'with from lhs to rhs 44a⟩
          (report 3 urn "expanded rhs is " (nice-type-structure rhs))
          (set! substructure-undefined-user-error #t)
          ; (3)
          (if (and (not (lookup-type rhs env)) (not (abstract-type? rhs)))
              (issue-error "Type " (nice-type-structure rhs)
                           " in declaration of " lhs " unknown")
              ; (4)
              (set! symbol-table-g
                    (cons (make-var-decl lhs rhs env)
                          symbol-table-g)))
          (set! substructure-undefined-user-error #f)
          ⟨output adjust: set res in variable declaration 66a⟩
          ))
         (set! cur-decl #f)
         res
         ) ; let ((urn))))

(define handle-local-var-decl
  (lambda (x type) ; where type \in { 'o, 'i, 'l }
    (set! cur-decl x)
    (let ((urn (string-append "#" (number->string unique-report-nr)))
          (res '()))
    (report 3 "reading local variable decl: " urn "\n" x "\ntype " type)

    ; (1)
    (if (and (find-var-lhs (strip-with (cadr x)) symbol-table-l)
             (not (is-st-constant? (cadr x))))
        (issue-error "Redeclaration of variable " (cadr x))
        ; (2)
        (let ((lhs (cadr x))
              (rhs (expand-aliases (preprocess-structure (caddr x))
                                type-be-decls-l))
                    (env type-be-decls-l))
                ⟨transfer 'with from lhs to rhs 44a⟩
                (report 3 urn "expanded rhs is " (nice-type-structure rhs))
                (set! substructure-undefined-user-error #t)
                ; (3)
                 (if (and (not (lookup-type rhs env))
                         (not (abstract-type? rhs)))
                    (issue-error "Type " (nice-type-structure rhs)
                                 " in declaration of " lhs " unknown")
                                               ; (4)
                    (begin
                      (set! symbol-table-l
                            (cons (make-var-decl lhs rhs env)
                                  symbol-table-l))
                      (if (eq? type 'i)
                          (set! symbol-table-i
                                (cons (car symbol-table-l)
                                      symbol-table-i)))
                      (if (eq? type 'o)
                          (set! symbol-table-o
                                (cons (car symbol-table-l)
                                      symbol-table-o)))
                      ⟨output adjust: set res in variable declaration 66a⟩
                    ))
                 (set! substructure-undefined-user-error #f)))
          (set! cur-decl #f)
          res
               ) ;let ((urn))))
      ◇
```

Macro defined by scraps 48, 51b, 52b.
Macro never referenced.

5.5 Algorithm Definitions and Prototypes

Algorithms and prototypes are entered into a global prototype-table. They establish the set of function identifiers which can be used within expressions and these identifiers can be overloaded. This feature is extensively exploited in SuchThat, since in the context of symbolic computation, especially operator symbols do have many meanings. Also SuchThat aims towards a translator which is able to determine which of two functions is the "more specialized" one with respect to input and output parameters and thus presumably the more efficient in terms of run time.

While "algorithms" give a complete description including the implementation, "prototypes" declare only input and output parameters, allowing the translator to check the correct calling of these functions, but postponing the implementation. [10]

⟨global variables 49⟩ ≡

```
      (define prototype-table '())
      ◇
```

Macro defined by scraps 24a, 27d, 43a, 46b, 49, 50a, 51a.
Macro never referenced.

In order to preprocess the found information as far as possible towards the needs of the overload resolution process, we use the following format for the entries into the table.

[10] This compares to the Pascal `forward` keyword and the C distinction between declaration and definition of functions.

(*name*[11] *signature internal-declarations visible-variable declarations body*)

and *signature* is

((*type* ...) [*type* ...])

for the input / output types respectively, the latter ones being possibly empty since some algorithms are just called for side effects.

The reasons to memorize the internal section and the visible (= local + global up to the algorithm) variable declarations in a designated place is that they may contain type parameters of the prototype which have to be accessed during instantiation of the algorithm. Also for the future purpose of instantiation we write out the body as well. It has been type checked with respect to the constraints of the template parameters. This is done in the algorithm `second-pass`. *body* is `#f` for a prototype.

The body will be communicated across the recursive file structure in the variable:

⟨global variables 50a⟩ ≡
```
(define define-step-list #f)
◇
```
Macro defined by scraps 24a, 27d, 43a, 46b, 49, 50a, 51a.
Macro never referenced.

As for the structure of an entry in the prototype table, we can define:

⟨access prototype table entries 50b⟩ ≡
```
(define make-prototype-entry
  (lambda (name signature internal-section visible-vars body)
    (list name signature internal-section visible-vars body)))

(define get-prototype-name
  (lambda (entry)
    (car entry)))

(define get-prototype-signature
  (lambda (entry)
    (cadr entry)))

(define get-prototype-internal
  (lambda (entry)
    (caddr entry)))

(define get-prototype-visible
  (lambda (entry)
    (cadddr entry)))

(define get-prototype-body
  (lambda (entry)
    (car (cddddr entry))))

(define make-signature
  (lambda (i-vars ovars)
    (cons i-vars o-vars)))
```

[11]Note that no unique names are computed so far, which can lead to ambiguities on the C^{++} level; however, “name mangling” does only make sense, if the translator resolves overloading to unambiguous expressions, which is not the case for this prototype.

```
(define get-signature-input
  (lambda (S)
    (car S)))

(define get-signature-output
  (lambda (S)
    (cdr S)))

(define get-prototype-input
  (lambda (entry)
    (get-signature-input (get-prototype-signature entry))))

(define get-prototype-output
  (lambda (entry)
    (get-signature-output (get-prototype-signature entry))))
◇
```

Macro never referenced.

Since the name, input, and output parameters of an algorithm are picked up in different places, we store them in the following variables.

⟨global variables 51a⟩ ≡

```
(define define-ilst '())
(define define-olst '())
(define define-name #f)
◇
```

Macro defined by scraps 24a, 27d, 43a, 46b, 49, 50a, 51a.
Macro never referenced.

which is set appropriately when we find the output list at the end of the body.

Reading the algorithm or prototype definition includes expanding the signature using aliases and storing the result in the normalized form indicated above.

⟨functions for special file parts 51b⟩ ≡

```
(define handle-algorithm
  (lambda (x)
    (report 3 "Handling prototype:\n" (caadr x))

    (let ((signature (cdadr x))
          (algo-name (caadr x)))
      (report 3 "Signature for " (caadr x) " is " signature)
         ⟨expand signature and enter into prototype table 52c⟩
         )
    ⟨clear local tables for next algorithm 52a⟩
    ;(list x)
    ⟨output adjust: throw away prototype 65c⟩
    ))
◇
```

Macro defined by scraps 48, 51b, 52b.
Macro never referenced.

The global references to the local symbol tables are no longer used since these are accessible through the references stored in the entry.

⟨clear local tables for next algorithm 52a⟩ ≡

```
; clear the define-ilst for next use
(set! define-ilst '())
(set! define-name 'invalid)
(set! define-olst '())

; clear reporting help
(set! cur-algorithm #f)

; clear local symbol table for next algorithm
(set! symbol-table-l '())
(set! symbol-table-i '())
(set! symbol-table-o '())

(set! type-be-decls-l type-be-decls-g)
(set! type-be-decls-d '())
◇
```

Macro referenced in scraps 20, 51b.

The handling of algorithms *with* bodies is only slightly more complicated since the list of output variables is found at the end of the body.

⟨functions for special file parts 52b⟩ ≡

```
(define handle-define-body
  (lambda (x) ; x is the algorithm-body
    (report 3 "Handling algorithm definition:\n" cur-algorithm)
    ; make list of input and output parameters
    ; just like for prototypes

    ; get parser regenerated return value into define-olst
    (let ((last (get-last-element (cddar(cddadr x)))))
      (set! define-olst (cond ((symbol? last) (list last))
                              ((and (pair? last) (eq? (car last) 'list))
                               (cdr last))
                              (#t '()))))
    (let* ((signature (if (null? define-olst)
                          define-ilst ; input parameters
                          (append define-ilst (list 'out) define-olst)))
           (algo-name define-name))

      (report 3 "Signature for " cur-algorithm " is " signature)
         ⟨expand signature and enter into prototype table 52c⟩
         )
    (list x)))
◇
```

Macro defined by scraps 48, 51b, 52b.
Macro never referenced.

Looking up parameters in the local symbol table gives us the expansion for the signatures. We enter the `var-decls` here.

⟨expand signature and enter into prototype table 52c⟩ ≡

```
 (report 3 "Input:  " (nice-var-decls symbol-table-i))
(report 3 "Output: " (nice-var-decls symbol-table-o))
```

```
    (report 3 "Local:  " (nice-var-decls symbol-table-l))
    (set! signature
          (let ((res '())
                (in-out 'i))
            ; transform old signature to new format
            (do ((s signature (if (and (pair? (cdr s))
                                       (eq? (cadr s) 'out))
                                  (begin
                                    (set! res (cons res '()))
                                    (set! in-out 'o)
                                    (cddr s))
                                  (cdr s))))
                ((null? s)
                 (if (eq? in-out 'o)
                     res
                     (cons res '()))) ; no output

              ; lookup variable declaration
              (let ((a (find-var-lhs (car s)
                                     (if (eq? in-out 'i)
                                         symbol-table-i
                                         symbol-table-o))))
                (if a
                    (set! res (append res (list a)))
                    (issue-error "Undefined parameter in algorithm " (caadr x)
                                 " : " (car s)))))))

    (report 3 "Expanded signature is \n"
            (nice-var-decls (car signature))
            "out\n"
            (nice-var-decls (cdr signature)))

    ; enter in prototype table with previously found body
    (report 3 "Entering to prototype table")
    (set! prototype-table
          (cons (make-prototype-entry algo-name signature
                                      type-be-decls-d
                                      (append symbol-table-l symbol-table-g)
                                      define-step-list)
                prototype-table))
    ◇
```

Macro referenced in scraps 51b, 52b.

6 Aliases in SuchThat

Aliases are introduced by *be* declarations. They simply serve as shortcuts or structural macros. The left and right hand side of a *be* declaration have perfectly the same meaning and are expanded in variable and *structure* declarations instantly, i.e. they *disappear* when they are used.

Concerning the relation between structures and aliases we see that aliases use structures as the base of expanding and on the other hand, bindable identifiers can be used in structures, for example to impose constraints upon a generic type used in their definition. These identifiers are *not* expanded.

The most important procedure is `expand-aliases` which has been used earlier.

6.1 The Semantics of Aliases

The algorithm to expand a structure S can be described as follows:

1. If the structure S is an identifier, do not expand it, since it may be bound. However, the S is replaced by its declaration to facilitate `merge-bindings`
2. Otherwise
   (a) `expand-aliases` in any substructure encountered. However, the *first-structure* will *not* be replaced if it is an identifier.
   Be careful in `'with` structures.
   (b) If the top-level-structure is known in the aliases via a structural match, expand it to the right hand side of its declaration, *carrying over* any substructures bound in the matching process.

This definition again shows aliases as shorthands:
If there is are declarations

Let I be a set. Let vector over I be vector(I).

then subsequently for example vector over fractionfield [integer] will be expanded into
vector(fractionfield [integer]).

6.2 Structural Matching for Aliases

A *structural match* is defined in the usual way with one important constraint:

An identifier in S may only be used *once*.

This enforces the semantics of a macro as opposed to a type, which requires consistent bindings instead of unique identifiers.

6.2.1 The Procedure `struct-match`

For the procedure `struct-match` we use the following algorithm: Given two structures S and T.

- If T is a bindable identifier, bind S to T if the constraint imposed by the declaration of T is fulfilled, otherwise return #f.
- If T is a structure, match all the substructures and check that any identifier is used only once.

Algorithm:	struct-match	
Input:	*A,B*	expanded structures, B a left hand side of of *be* declaration with the imposed constraints.
	Aenv	contains declarations of subtypes of A.
Output:	*bind*	A set of bindings (see `make-binding`) of bindable identifiers in B to substructures of A if the match was possible, otherwise #f.

⟨structural matching 54⟩ ≡

```
(define struct-match
  (lambda (A B Aenv)
    (let ((urn (string-append "#" (number->string unique-report-nr))))
    (report 9 "struct-match "
            (nice-type-structure A) " |= " (nice-type-structure B) "\nin"
            (nice-decls Aenv 'be))
    (cond  ((bindable-identifier? B)
            (let ((subb (simplies-explicit? A Aenv
                                            (get-be-decl-rhs B)
                                            (get-be-decl-environment B))))
              (if (not subb)
                  (begin
                    ; this might be useful at some point
                    ;(issue-warning "Rejected binding because of constraints\n"
                    ;(nice-be-decl B) " |-> "
                    ;(nice-type-structure A))
                    #f)
                  (list (make-binding B A Aenv subb)))))

           ((and (pair? A) (not (memq (car A) known-structure-tags))
                 (not (bindable-identifier? A)))
            (report 0 "Ill formed structure in match: "
                    (nice-type-structure A))
            (error "internal error"))
           ((and (pair? B) (not (memq (car B) known-structure-tags))
                 (not (bindable-identifier? B)))
            (report 0 "Ill formed structure in match: "
                    (nice-type-structure B))
            (error "internal error"))

           ; structural matching
           ((and (pair? A) (pair? B)
                 (memq (car A) known-structure-tags)
                 (eq? (car A) (car B))
                 (pair? (cdr A)) (pair? (cdr B))
                 (or (and (symbol? (cadr A)) (symbol? (cadr B))
                          (eq? (cadr A) (cadr B)))
                     (and (pair? (cadr A)) (pair? (cadr B)))))
            (report 9 urn "matching substructures")
           ; match substructures
            (do ((aa (if (symbol? (cadr A)) (cddr A) (cdr A)) (cdr aa))
                 (bb (if (symbol? (cadr B)) (cddr B) (cdr B)) (cdr bb))
                 (res '() res))
                ((or (not res)
                     (null? aa) (null? bb))
                 (report 9 urn "result is " (nice-bindlist res))
                 (if (and (null? aa) (null? bb))
                       res
                       #f))
              (set! res (struct-match-merge res
                                            (struct-match (car aa) (car bb)
                                                          Aenv)))))
           (#t #f)))))

(define struct-match-merge
  (lambda (sigma tau)
    (cond  ((or (not sigma) (not tau)) #f)
           ((null? sigma) tau)
           ((find-binding (car sigma) tau)
                (issue-error "Constraint upon macros violated.\n"
                             "Identifier " (caar sigma) " used twice")
                #f
                )
               (#t (let ((r (struct-match-merge (cdr sigma) tau)))
                     (if r
                         (cons (car sigma) r)
                         #f))))))
;(trace struct-match-merge)
◇
```

Macro never referenced.

6.3 Expanding Aliases to Structures

The following algorithm is very much alike the instantiation case which for $S \models T$ uses the declaration, which S refers to.

Algorithm:	expand-aliases	
Input:	S	a pre processed structure
	aliases	an environment, of which only the *be* declarations will be used.
Output:	S'	where all the aliases in *aliases* applicable to S or substructures of S have been expanded.

⟨expanding aliases 56⟩ ≡

```
(define expand-aliases
  (lambda (s aliases)
    (let ((urn (string-append "#" (number->string unique-report-nr))))
    (report 9 "expand-aliases: " (nice-type-structure s)
            "\nin\n" (nice-decls aliases 'be))
    (cond ((symbol? s)
           (let ((lu (lookup-alias s aliases)))
             (if lu
                 (car lu) ; (cdr lu) = '()
                 s
                 )))
          ((pair? s)
           ; expand all substructures
           (set! s
                 (cond ((attributed? s)
                        (cons (car s)
                              (cons (expand-aliases (cadr s) aliases)
                                    (cddr s))))
                       ((symbol? (cadr s))
                        (cons (car s)
                              (cons (cadr s)
                                    (map (bind2nd expand-aliases aliases)
                                         (cddr s)))))
                       (#t (cons (car s)
                                 (map (bind2nd expand-aliases aliases)
                                      (cdr s))))))
           (report 9 "after expanding substructures " urn "\n"
                   (nice-type-structure s))
           (let ((l (lookup-alias s aliases)))
              (if l
                  (begin
                    (set! s (apply-bindings (cdr l) (get-be-decl-rhs (car l))))
                    (report 9 "top-level alias found " urn))))
            (report 9 "after expanding top-level alias " urn "\n"
                    (nice-type-structure s))
            s
            )
           (#t s))) ; let ((urn))))
◇
```

Macro defined by scraps 56, 57.
Macro never referenced.

6.4 Lookup for Aliases in Tables

The lookup function is implemented the obvious way: We walk the environment and return the first alias declaration whose left hand side does structurally match the given S with all constraints fulfilled.

Algorithm:	lookup-alias	
Input:	S	a structure where all the substructures have been expanded
	aliases	an environment
Output:	*out*	A pair of the first *be* declaration in the list of aliases that is applicable to S and the bindings found in the structural matching used. If no such match is obtained for any declaration in *aliases*, out = #f.

⟨expanding aliases 57⟩ ≡

```
(define lookup-alias
  (lambda (s aliases)
    (let ((urn (string-append "#" (number->string unique-report-nr))))
    (report 9 "lookup-alias for " (nice-type-structure s)
            "\nin\n" (nice-decls aliases 'be))
    (do ((a aliases (cdr a))
         (res #f res))
        ((or (null? a)
             res)
         (report 9 urn " finished lookup-alias with result\n"
                 (if res
                     (nice-decl (car res) 'both)
                     #f
                     ))
         res)
      (cond ((not (eq? (get-decl-tag (car a)) 'be)))
            ((symbol? s)
             (if (eq? s (get-be-decl-lhs (car a)))
                 (set! res (cons (car a) '()))))
            ((and (pair? s) (pair? (get-be-decl-lhs (car a))))
             (set! res (cons (car a) (struct-match s
                                                   (get-be-decl-lhs (car a))
                                                   aliases)))
             (if (not (cdr res))
                 (set! res #f)))
            (#t #f)))) ; let ((urn))))
◇
```

Macro defined by scraps 56, 57.
Macro never referenced.

7 An Interface to stgentz

Now we have to answer the question

structure-implication?

We will do this by using the tool for algebraic resolution, STGENTZ by Rüdiger Loos and Sibylle Schupp ([11]). However, since we expect the same questions to arise more than once, we also provide a cache for implications already found to be #t or #f.[12]

7.1 The Function `structure-implication?`

For statistical purposes, we define a counter for the questions, which may later be compared to the counter for cache hits.

The procedure itself contains three steps:

1. If the structures are `equal?` return #t.
2. Lookup the implication cache.
3. Forward the question to STGENTZ.

The third step is not yet contained in the following procedure but the necessary changes can be easily accomplished.

⟨the `structure-implication?` predicate 58⟩ ≡

```
(define calls-to-structure-implication 0)

(define structure-implication?
  (lambda (A B)
    (set! calls-to-structure-implication
          (+ 1 calls-to-structure-implication))
    (report 1 "Question to structure-implication? :\n"
            A " => " B)

    (cond ((equal? A B) (report 1 "Syntactic equivalence") #t)
          (#t (let ((res (lookup-implication-cache A B)))
                (if (null? res) ; not found in cache
                    (begin
                      (report 1 "Implication not found; assuming #f")
                      (issue-warning "Implication not found\n"
                                     (list A  B "." #t) "\n"
                                     ))
                    (report 1 "Answer is: " res))
                   (and res (not (null? res))))))))
◇
```

Macro never referenced.

[12]For testing of overload resolution, we use this cache as a single basis for structure implications. We describe however the general measures to be taken to connect to STGENTZ.

7.2 Connect to stgentz

We give an outline of the procedure `stgentz?` by means of literate programming without filling out the details at this time.

⟨outline of an interface to STGENTZ 59a⟩ ≡
```
(define stgentz?
  (lambda (A B)
    ⟨transcribe A and B to the STGENTZ syntax ?⟩
    ⟨call the entry function of the tool ?⟩
    ))
◇
```
Macro never referenced.

Although in this module we have used the structures almost as they were delivered by the parser, this was enough to compare two structures, since if they look the same in the source code, the intermediate representation will also look alike.

STGENTZ however needs in a sense more structural information. It requires the substructures to be assigned to a structure in a linear form, i.e. we have the structure's name plus a set of attributes plus a set of parameters to describe the structure:

`(from (to map field) set)` ⇒ `map{from:set}{to:field}`
and
`(to (from map set) field)` ⇒ `map{from:set}{to:field}`

This assignment partially undoes the work of the parser in that it recovers the linear structure, which the user sees when typing the source code from the recursive intermediate representation. This transformation thus depends on details of the grammar used to generate the parser, such as left or right associativity.

Also, there must be some way of distinguishing the different structure declarations used, although we needn't fear loopholes, since the constraints have been checked in `fully-expand`.

7.3 Caching Results of Structure Implications

Since programs typically deal with only few types it is very likely that the same structure implications have to be decided several times.

Be aware though, that the cache must be cleaned upon a database change. This might occur in a later version of the translator, when declarations and type checking are done in the same pass. Right now it is of no importance.

⟨A cache for implications 59b⟩ ≡
```
(define found-in-cache 0)

(define cache-file (open-input-file "cache.imp"))
(define implication-cache (read cache-file))
(report 1 "Using cache:\n" implication-cache)

(define lookup-implication-cache
  (lambda (A B)
    (let ((res '()))
      (do ((c implication-cache (cdr c)))
          ((or (null? c)
               (not (null? res)))
           (if (not (null? res))
                    (set! found-in-cache (+ 1 found-in-cache)))
                res
                )
            (if (and (equal? A (caar c))
                     (equal? B (cadar c)))
                (set! res (cddar c)))))))
(define cache-implication
  (lambda (A B holds?)
    (set! implication-cache
          (cons (cons A (cons B holds?))
                implication-cache))))
◇
```

Macro never referenced.

8 Instantiation of Structures

In the previous section, the conceptual design of an interface to STGENTZ has been presented. However STGENTZ lacks a notion of substitution and bindings, which makes it necessary to specify and implement a separate procedure. At this point, the function `instantiate` performs this task. Since it is an experimental implementation and the details are likely to change, we do not give the code here.

However, we want to both motivate and describe the overall design by particular examples demonstrating situations `instantiate` has to deal with. These examples also give a guideline for what has to be accomplished in incorporating the idea of bindings into the Gentzen system.

8.1 The Notion of `instantiation`

Besides intuition, the implementation has to be based on the definitions given in [13] and be consistent with the calculus presented. However the interpretation of the structure declaration (see below) is not included in these definitions and the implementation cannot follow previous intentions here.

The main idea of `instantiation` is a recursive structural matching between two structures S and T. Type-variables are introduced using the `Let...be...` declaration where the left hand side is a symbol. The type-variables occurring in T may be bound corresponding subexpressions in S iff the subexpression itself instantiates the right hand side of the type-variable declaration. This effectively introduces constraints on bindings and ensures the check of preconditions in algorithm calls. As always in the situation of pattern matching, seen as one-sided unification, we have to pay attention to the questions of

- Consistency
- Occurrence checking

Examples:

Let R be Ring.
Let P be Polynomial over Ring.
a) Matrix over P
b) Matrix over Polynomial of Z/4Z

Now structure b) is an instance of structure a) by the following reasoning:

1. A `Matrix` matches a `Matrix` in case the element types match.

2. P is a `Polynomial`, so it matches the `Polynomial` in b) if the coefficients match.

3. $Z/4Z$ is a `Ring`

The bindings we find are:

```
P ↦ Polynomial over Z/4Z
R ↦ Z/4Z
```

Consistency is mainly needed for function calls:

```
Algorithm: q := operator *(a,p)
           Let C be Ring;
           Let P be Polynomial over C.
Input:     $a \in$ C;
           $p \in$ P.
Output: $q \in$ P.
```

We have to make sure that we use the right coefficient domain for the polynomials. Another example constructing homogenous lists can be found in section 2.2. A few remarks conclude this subsection:

- The identifiers, which can be bound to some structure, are introduced by `Let...be...` declarations.
- Global declarations of this kind are treated as if copied to the internal section of every algorithm that refers to them.
- The order in which the actual parameters are matched against the formal parameters is not significant, since *consistency* is a property of the (set) union of all sets bindings found in the individual matchings.
- *Attributes* are treated the following way (we use the internal representation for easier display): Can we instantiate

  ```
  (with (sequence over Ring) finite)
  by
  (with (sequence over Integer) finite nonempty)
  ```

  The answer is (see [13]): Yes we can, as we can instantiate (without any bindings) sequence over Ring by sequence over Integer since the integers form a ring and furthermore the set {finite} is a subset of {finite, nonempty}.

8.2 Semantics of Structure Declarations

In section 5.3 we have introduced a declaration for structures. Examples are:

a) Structure Integer is implemented;

b) Structure Array(Set) is implemented;

c) Structure Module over Ring is abstract;

d) Let S be SemiGroup;
Structure Vector over S is Array(S).

Let's go through these declarations: *a*) states, that a structure Integer exists without any parameters. The word implemented is *not* a keyword but simply a remark to the human reader.

$b - d$ introduce 3 parameterized structures: An array, which can be used with any set, the concept of a module, which is defined over a ring only and finally the concept of a vector, which is defined in terms of an array, but unlike this data structure, requires its parameter to be a semi group.

Suppose now that we want to call a function, whose formal parameter is a Array(Integer) with a *Vector over Natural*. Obviously, they do not match directly, but looking at the declaration of vector, we find that when binding S to Natural, which is possible since natural numbers are a semi group, our actual parameter refers to declaration *d*). Substituting this binding on the right hand side gives us that actually a Vector over Natural is equal to a Array(Natural). Now we can match Array(Integer) with Array(Natural) since the natural numbers are simply the positive (or non-negative) integers and we get a legal function call.

For the current implementation, the instantiation procedure requires structures, which are not simply identifiers, to be declared before use. When two structures are matched, they need not only correspond syntactically, but also need to refer to the same declaration. This restriction has been introduced for the following reason: In declaration *d*) above, we require vectors to be constructed over a SemiGroup. If a local declaration of an algorithm introduces another vector structure with different constraint, we want to keep these structures apart.

8.3 Weaving Implication and Instantiation

Since a merging of structure implication and instantiation has not been accomplished so far, we interconnect them as closely as possible by the procedure `simplies-explicit?` which first tries to instantiate two structures and upon failure asks a decision from the structure implication.

The instantiation procedure is in general more restrictive and gives more precise results, so this two step procedure is justified.

Algorithm:	simplies-explicit?	
Input:	*x,y*	structures with aliases expanded down to the bindable identifiers
	x-env,y-env	the environments describing the possible substructures of x and y
Output:	*out*	• #f if not $x \Rightarrow_S y$ • `'()` if the structure implication has been used and returned #t • The binding list that `instantiate` returned if it succeeded in matching the structures.

⟨the predicate simplies? 62⟩ ≡

```
(define simplies-explicit?
  (lambda (x x-env y y-env)
    (let ((res (instantiate x x-env y y-env))
          (fex '())
          (fey '()))
      (if res
          res
          (begin
            (set! fex (fully-expand x x-env))
```

```
                (set! fey (fully-expand y y-env))
                (report 1 "simplies? using structure-implication?\n"
                        fex " => " fey)
                (if (structure-implication? fex fey)
                    (begin
                      (issue-warning "simplies-explicit? used structure-implication?\n"
                                     fex " => " fey
                                     "\n=> no bindings found")
                      '())
                    #f
                    ))))))
```

◇

Macro never referenced.

8.4 Error Messages

With the introduction in this section we are now able to give the user of ST2CPP hints from where possible errors emerge.

"type of right hand side unknown"
A variable, structure or alias is declared in terms of an unknown structure. Parameterized structures need to be defined before they are used.

"Constraints upon alias lhs violated"
One of the bindings which occurred when expanding an alias was not legal.

"Redeclaration of variable " x
Variables cannot be overloaded.

"Type " S **" in declaration of "** x **" unknown"**
See **"type of right hand side unknown"**

"Left hand side of assignment unknown"
The variable that is to be assigned a value is not declared.

"Undefined parameter in algorithm"
All input and output parameters in an algorithm header have to be declared in the input and output section resp.

"Couldn't find prototypes with name " name **" and "** n **"input parameter(s)"** Before matching types of parameters, the set of all prototypes with the right name and number of parameters is collected. Probably the name is misspelled.

"No matching prototype found for expression"
Check the preceding warnings and errors for hints where an instantiation might have failed.

"non-supported number format"
SuchThat assumes all constants to be declared before use. For convenience, the prototype assumes the default interpretation for integer literals. If they are positive, also a *natural* interpretation is offered.

"Undeclared constant / variable"
All constants and variables except integers must be declared.

"Output variable not defined"
In a function call the type of an output parameter of the called algorithm could not be determined. This message is preceded by an **"Undefined parameter in algorithm"**

"No prototypes matching output signature"

"No prototypes matching desired type"
Certain locations of expressions in the input require a particular type to be returned, for example boolean expressions in while or if statements or integer expressions for for loops. None of the possible interpretations in terms of prototypes returned a structure implying the desired one. This mechanism is also used for assignments.

"Declared variable of type " T **" found, expected "** S
The variable used in a for statement was not declared.

"Variable " v " of wrong type"
In for statements, only variables with a structure implied by Integer may be used.

"Redeclaration in " declaration
Right now structures may not be redeclared. A structure, which can be instantiated by the left hand side has been given before.

"Constraint upon macros violated. Identifier used twice"
See section 5.1.

"Undefined structure used"
During instantiation a substructure was undeclared.

The warnings issued are self-explanatory:

"Rejected binding because of constraints" One of the bindings (see above) could not be accomplished because the limitation introduced by the be declaration of the identifier was not fulfilled.

"Implication not found" The translator could prove the implication neither right nor wrong based on its knowledge base `cache.imp`. Enter the given implication to the file. If the implication does not hold, change the #t to #f.

"Instantiation failed because of missing attributes"
When instantiating attributed structures like `(with T B)` by `(with S A)` then according to [13] it is sufficient to demand that S must be an instance of T and B be a subset of identifiers of A. The second condition is violated.

"Inconsistent binding occured during merge" If a structure contains the same identifier, which is introduced in a be declaration in two substructures, these bindings must be consistent.

"Inconsistent binding occured extracting" If the same identifier is used in different input parameters to an algorithm, the bindings must be consistent.

"simplies-explicit? used structure-implication?" This condition issues a warning because bindings have not taken place, which might mean that inconsistencies within a function call are not discovered. However, the instantiation can discover bindings in one subpart of a structural match and use the structure implication in another. This means that probably the warning can be ignored.

"Output signature mismatch" A prototype could not be used because of its output parameters, i.e. the actual parameters matched the formal ones, but the return values could not be stored in the given variable because the returned structure did not imply the variables declaration. This can only happen if the function call is of the form

```
f(x ; y)
```

9 The Connection to scm2cpp

This section describes the modifications done to the output of the type checker in order to match the specifications of the backend and the customizations made inside of the backend.

9.1 Adjusting the Output

Here is the place where we collect all the changes made to the output of the typechecker. All of these will be done using STWEB macros such that when reading the code in the former sections the reader may quickly retrieve the relevant information by looking at the crossreferences given by STWEB.

Most of the following macros rename tags, where the backend expects something else than is currently there or throw away information it cannot deal with, such as declarations of structures and aliases.

⟨output adjust: output the original expression 65a⟩ ≡

```
(list e)◇
```

Macro never referenced.

⟨output adjust: throw away internal declarations 65b⟩ ≡

```
,cut-out◇
```

Macro referenced in scrap 22a.

⟨output adjust: throw away prototype 65c⟩ ≡

```
'()◇
```

Macro referenced in scrap 51b.

⟨output adjust: rename tag to st-local 65d⟩ ≡

```
,(lambda (x)
   (list (cons 'st-local (cdr x))))
◇
```

Macro referenced in scrap 22a.

⟨output adjust: rename tag to st-input 65e⟩ ≡

```
,(lambda (x)
   (list (cons 'st-input (cdr x))))
◇
```

Macro referenced in scrap 22a.

⟨output adjust: rename tag to st-output 65f⟩ ≡

```
,(lambda (x)
   (list (cons 'st-output (cdr x))))
◇
```

Macro referenced in scrap 22a.

⟨output adjust: set res in variable declaration 66a⟩ ≡

```
(set! res
      (if (not (is-st-constant? lhs))
          (list (cons lhs (postprocess-structure
                           (convert-bindable-to-ids
                            (expand-structure-decls
                             (strip-attributes rhs)
                             env)))))
          '()))
◇
```

Macro referenced in scrap 48.

Of course we have to undo the changes by `preprocess-structure`

⟨inverse operation of preprocess structure 66b⟩ ≡

```
(define postprocess-structure
  (lambda (T)
    (cond ((symbol? T) T)
          ((bindable-identifier? T) T)
          ((and (pair? T) (eq? (car T) 'app_par))
           (map postprocess-structure
                (cdr T)))
          ((and (pair? T) (memq (car T) known-structure-tags))
           (cons (car T)
                 (map postprocess-structure
                      (cdr T))))
          (#t (report 0 "Ill formed structure in post-processing " T)
              (error "internal error")))))
;(trace postprocess-structure)
◇
```

Macro never referenced.

The backend does not know bindings and the special role of identifiers here. It simply treats any identifier it does not know as a template parameter.

⟨output auxiliary functions 66c⟩ ≡

```
(define convert-bindable-to-ids
  (lambda (T)
    (cond ((symbol? T) T)
          ((bindable-identifier? T)
           (get-be-decl-lhs T))
          ((and (pair? T) (memq (car T) known-structure-tags))
           (do ((res (list (car T)) res)
                (l (cdr T) (cdr l)))
               ((null? l) (reverse res))
             (set! res (cons (convert-bindable-to-ids (car l))
                             res))))
          (#t (report 0 "Ill formed structure in convert-bindable-to-ids\n"
                      T)
              (error "internal error")))))
;(trace convert-bindable-to-ids)
◇
```

Macro defined by scraps 66c, 67b.
Macro never referenced.

Structure declarations are useful to express a concept and its constraints. The backend does not need to know them. By expanding the structure declarations as if they were macros, we do output storable types in case no abstract structures have been used.

⟨expanding structure declarations 67a⟩ ≡
```
(define expand-structure-decls
  (lambda (type-struct env)
    (if (not (bindable-identifier? type-struct))
        (let ((lu (lookup-type type-struct env)))
          (if (and lu (get-structure-decl-lookup (car lu)))
              (expand-structure-decls
               (apply-bindings (cdr lu)
                               (get-structure-decl-rhs (car lu)))
               (get-structure-decl-environment (car lu)))
              ; else
              (if (symbol? type-struct)
                  type-struct
                  (if (symbol? (cadr type-struct))
                      (append (list (car type-struct) (cadr type-struct))
                              (map (bind2nd expand-structure-decls env)
                                   (cddr type-struct)))
                      (append (list (car type-struct))
                              (map (bind2nd expand-structure-decls env)
                                   (cdr type-struct)))))))
        type-struct
        )))
;(trace expand-structure-decls)
◇
```
Macro never referenced.

Since attributes, similar to structure declarations, are not expressible in C^{++}, we have to dispose of them for the final output.

⟨output auxiliary functions 67b⟩ ≡
```
(define strip-attributes
  (lambda (type)
    (cond ((bindable-identifier? type) type)
          ((symbol? type) type)
          ((attributed? type)
           (strip-attributes (cadr type)))
          (#t (if (symbol? (cadr type))
                  (append (list (car type) (cadr type))
                          (map strip-attributes
                               (cddr type)))
                  (append (list (car type))
                          (map strip-attributes
                               (cdr type))))))))
;(trace strip-attributes)
◇
```
Macro defined by scraps 66c, 67b.
Macro never referenced.

⟨output adjust: throw away "be" declarations 67c⟩ ≡
```
'()◇
```

Macro referenced in scraps 39b, 40b.

⟨output adjust: rename tag and delete empty "global" 68a⟩ ≡
```
,(lambda (x)
   (if (null? (cdr x))
       '()
       (list (cons "st-global" (cdr x)))))
```
◇

Macro referenced in scrap 20.

⟨output adjust: throw away "type" declarations 68b⟩ ≡
```
'()
```
◇

Macro referenced in scraps 43b, 45a.

⟨output adjust: parser generated return value 68c⟩ ≡
```
(if (not (list? x))
    (list (list 'return x))
    '() ; C++ can't return multiple values
    )
```
◇

Macro referenced in scraps 22b, 24b.

The backend does not expect the file to contain one list but a sequence of SCHEME objects.

⟨output adjust: write output as sequence 68d⟩ ≡
```
(do ((r result (cdr r)))
      ((null? r))
    (display (car r) output-port))
```
◇

Macro referenced in scrap 12.

⟨output adjust: multiple return values 68e⟩ ≡
```
(set! e1 (append (list (cadr e)) input output))
```
◇

Macro never referenced.

⟨output adjust: set signature in ret 68f⟩ ≡
```
(set! ret
      (cons 'define
            (cons (append (list (caadr x)) ; name
                          define-ilst
                          define-olst)
                  (cddr x))))
```
◇

Macro referenced in scrap 20.

10 Acknowledgments

We like to acknowledge all who were members of the core SUCHTHAT groups at RPI and in Tübingen at the time of the work on ST2CPP. In alphabetical order this is Stephen A. Cerniglia, Albrecht Haug, Uwe Kreppel, Gor Nishanov, Christoph Schwarzweller, Quincy Stokes, Sebastian Wedeniwski, Roland Weiss.

References

[1] Revised^4 Report on the Algorithmic Language Scheme, 1991.

[2] Herold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques, and tools.* Reading, Mass. : Addison-Wesley Pub. Co., 1986.

[4] T. P. Baker. A one-pass algorithm for overload resolution in Ada. *ACM Transactions on Programming Languages and Sytems*, 4(4):601–614, October 1982.

[5] Preston Briggs. nuweb a simple literate programming tool. `preston@cs.rice.edu`, May 1989.

[6] Harald Ganziger and Knuth Ripken. Operator identification in Ada: Formal specification, complexity, and concrete implementation. *ACM SIGPLAN Notices*, pages 30–42, Feb. 1980.

[7] Holger Gast. With scheme from SuchThat to c++. Studienarbeit, 1997.

[8] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, 1986.

[9] Rüdiger Loos. A SuchThat Translator to Scheme, 1997. Internal Report.

[10] Rüdiger Loos and Sibylle Schupp. Associating SuchThat Structures with their Attributes. Internal Report, 1997.

[11] Rüdiger Loos and Sibylle Schupp. An implementation of structure implication sequents. Internal Report, 1997.

[12] Sibylle Schupp. *Generic Programming – SuchThat One Can Build an Algebraic Library.* PhD thesis, University of Tübingen, 1996.

[13] Sibylle Schupp. Deciding structure implications. 1997.

[14] Sibylle Schupp. Processing SuchThat declarations. Internal Report, May 1997.

[15] Alexander Stepanov and Meng Lee. *The Standard Template Library.* Hewlett-Packard Company, Palo Alto, 1995.

[16] Bjarne Stroustrup. *The C++ Programming Language.* Addidson-Wesley Publishing Company, 2nd edition, 1991.

[17] Holger Szillat. Finding the SuchThat-Expressions to Statically type-check. Studienarbeit, 1997.

[18] Roland Weiss. ScmToCpp : A configurable, intelligent back end for SuchThat. Diplomarbeit, 1997.

A The Translator st2cpp

The following script for the BOURNE SHELL basically calls every part with the output of the previous stage. It integrates the modules into a translator from SuchThat to C^{++}.
The variables `SCM` and *STWEB*

`"st2cpp"` 69 ≡

```
#!/bin/sh

SCM=scm
STWEB=~/bin/stweb

PARAM=$1
if test $1 = "-p"
then shift
fi
: sh $STWEB -t sttype.w
if test -z "$1";
then echo Usage is st2cpp [-p] '"name"';
elif test ! -r $1.sth;
then echo Input file '"'$1.sth'"' not found;
else
echo Parsing SuchThat input $1.sth "->" $1.scm;
rm -f $1.scm
if st2scm $1.sth > $1.scm
then
if test $PARAM = "-p"
then
echo Aborting translation with -p flag
exit
else
echo Attribute association $1.scm "->" $1.staa $1.pp1;
rm -f $1.staa $1.pp1
cat $1.scm|$SCM  sta1.scm  sta1main.scm> $1.staa;
cat $1.staa|$SCM   pretty.scm > $1.pp1;
:
echo Typecheck $1.staa "->" \
$1.type $1.pp2 $1.imp sttype.err sttype.log;
: remove stale files
rm -f sttype.log sttype.err sttype.out $1.type $1.pp2;
cat $1.staa|$SCM   sttype.scm;
: Process some log messages;
grep -A 1 Quest sttype.log > $1.imp;
grep -c Quest sttype.log >> $1.imp;
: Pretty print the output;
cat sttype.out|$SCM   pretty.scm > $1.type;
cat $1.type|$SCM   pretty.scm >  $1.pp2;
:
echo sttype.err contains `grep warning: sttype.err|wc -l` \
warnings and `grep error: sttype.err|wc -l` errors.;
:
if test ! `grep error sttype.err|wc -l` -eq 0;
then echo "Compilation stopped because of errors";
else
echo Generating C++ output $1.type "->" $1.cpp
scm2cpp -ac $1.type
mv $1.type.cpp $1.cpp
fi
fi
else echo Parse error;
fi
fi
```

◇

B Examples

Some of the examples that ST2CPP has been used on, are given here. They roughly correspond to those used in [18] augmented with the function declarations necessary to type check the bodies. Furthermore, we include demonstrations of the instantiation procedure including attributes and parameterized structures.

B.1 Computing the GCD

Please note the the *warnings* indicated by ST2CPP are not all relevant. As described in section 2 they contain hints to possible explanations of occuring errors and can be ignored if no errors are signaled.

"gcd.sth" 71 ≡

```
Global: Let I be integral domain;
        Structure AmpleSet of I is I.

Algorithm: t:=operator =(x,y)
        Let S be Set.
Input:  x,y ∈ S.
Output: t ∈ bool. ||

Algorithm: t:=operator !=(x,y)
        Let S be Set.
Input:  x,y ∈ S.
Output: t ∈ bool. ||

Global: let E be EuclideanRing;
        let Amp be AmpleSet of E.

Algorithm: QR(x,y;q,r)
Input:     x,y ∈ E such that y ≠ 0.
Output:    q,r ∈ E such that x = q*y+r and (r = 0 or deg(r) < deg(y)). ||

Global: let I be integral domain;
        let Amp be AmpleSet of I.

Algorithm: y := NF(x)
Input:     x ∈ I.
Output:    y ∈ I. ||

Algorithm: d:=deg(e)
Input: e ∈ E.
Output: d ∈ Integer. ||

Global: 0 ∈ E. // with  0 is additive identity of E.

Algorithm: c := GCD(a,b)
Input:     a,b ∈ E.
Output:    c ∈ E. // such that //c ∈ Amp and c = gcd(a,b).
Local: u,v,s,t ∈ E.
(1) //Initialization
    u := a;
    v := b.
(2) //a = 0
    if u = 0 then {c := NF(v); return}.
(3) //Loop
    while v ≠ 0 do
      { QR(u,v;s,t);
        u := v;
        v := t }.
(4) //Normalization
    c := NF(u)   ||

Algorithm: main()
Local: x ∈ Integer.
(1) x:=gcd(35,14);
    print x;
    newline  ||
◇
```

For the C^{++} output please note that we consider the function `nf()` a library function, since it implements a normalization for the given domain. Furthermore the use of

```
e (*c) = new e;
```

does not cause memory leaks because we work in a garbed-collected environment. The C^{++} output generated was

⟨GCD C^{++} output 72⟩ ≡

```
// File created by scm2cpp
// gcd.type -> gcd.type.cpp

#include <iostream.h>

template<class e>
e & gcd(const e & a, const e & b) {
  e u;
  e v;
  e s;
  e t;
  e (*c) = new e;
  u = a;
  v = b;
  if ((u == 0)) {
    (*c) = nf(v);
    return (*c);
  }
  while ((v != 0)) {
    qr(u, v, s, t);
    u = v;
    v = t;
  }
  (*c) = nf(u);
  return (*c);
}
void main() {
  int x;
  x = gcd(35, 14);
}
// --- EOF by scm2cpp ---
◇
```

Macro never referenced.

B.2 Henrici-Brown Addition for Fraction Fields

The second example is the Henrici-Brown algorithm for addition in fractionfields of gcd domains.

"hb+.sth" 73 ≡

```
// Prefixes to specify details
Global: Let I be a gcddomain;
        Structure FractionField of I is abstract;
        Structure AmpleSet of I is abstract.

Algorithm: t:=operator !=(x,y)
        Let S be a set.
Input: x,y ∈ S.
Output: t ∈ bool. ||

Algorithm: t:=operator =(x,y)
        Let S be a set.
Input: x,y ∈ S.
Output: t ∈ bool. ||

Algorithm: z:=operator and(x,y)
Input: x,y ∈ bool.
Output: z ∈ bool. ||

Algorithm: z:=operator +(x,y)
        Let J be a gcddomain.
Input:  x,y ∈ J.
Output: z ∈ J. ||

Algorithm: z:=operator *(x,y)
        Let J be a gcddomain.
Input:  x,y ∈ J.
Output: z ∈ J. ||

// original example
Global: Let I be a gcddomain;

        Let Q be a FractionField of I;

        0 ∈ I with 0 is normalized;
        1 ∈ I with 1 is normalized;

        0 ∈ Q with 0 is normalized;
        1 ∈ Q with 1 is normalized;

        Let A be AmpleSet of I.

Algorithm: r1 := num(r)
```

```
Input:     $r \in Q$.
Output:    $r1 \in I$. // $r1$ the numerator of $r$
||

Algorithm: $r2$ := denom($r$)
Input:     $r \in Q$.
Output:    $r2 \in I$. // $r2$ the denominator of $r$
||

Algorithm: $r$ := fract($r1$, $r2$)
Input:     $r1, r2 \in I$ with $r2 \neq 0$.
Output:    $r \in Q$ with $r$ = $r1/r2$ ,  $r$ is normalized.
||

Algorithm: $r$ := operator/($r1$, $r2$)
Input:     $r1, r2 \in I$ such that $r2 \neq 0$, $r2$ %  $r1$ = 0.
Output:    $r \in I$ with $r$ = $r1/r2$.
||

Algorithm: $r$ := gcd($r1$, $r2$)
Input:     $r1, r2 \in I$.
Output:    $r \in I$. //with $r$ is the g.c.d. of $r1$ and $r2$.
||

Algorithm: $t$ := operator+($r$, $s$)
Input:     $r, s \in Q$ such that r is normalized, s is normalized.
Output:    $t \in Q$ such that $t = r + s$, $t$ is normalized.
Local:     r1,r2,s1,s2,t1,t2,$r2^\star$,$s2^\star$,
                $t1^\star$,$t2^\star$, d,e $\in$ I.
(1) // $r$ = 0 or $s$ = 0 ?
    if $r$ = 0 then { $t$ := $s$; return };
    if $s$ = 0 then { $t$ := $r$; return }.
(2) // get numerators and demominators
    $r1$ := num($r$); $r2$ := denom($r$);
    $s1$ := num($s$); $s2$ := denom($s$).
(3) // $r$ and $s \in I$
    if ($r2$ = 1 /\ $s2$ = 1) then { $t$ := fract($r1 + s1$, 1); return }.
(4) //$r$ or $s \in I$
    if $r2$ = 1 then { $t$ := fract($r1 \cdot s2 + s1$, $s2$); return };
    if $s2$ = 1 then { $t$ := fract($s1 \cdot r2 + r1$, $r2$); return }.
(5) //general case
    $d$ := gcd($r2$, $s2$);
    if $d$ = 1 then { $t$ := fract($r1 \cdot s2 + r2 \cdot s1$, $r2 \cdot s2$); return };
    if $d \neq$ 1 then {
      $r2^\star$ := $r2/d$; $s2^\star$ := $s2/d$;
      $t1$ := $r1 \cdot s2^\star + s1 \cdot r2^\star$; $t2$ := $r2 \cdot s2^\star$;
      if $t1$ = 0 then { $t$ := 0; return };
      if $t1 \neq$ 0 then {
        $e$ := gcd($t1$, $d$);
        $t1^\star$ := $t1/e$; $t2^\star$ := $t2/e$;
        $t$ := fract($t1^\star$, $t2^\star$) }} ||
◇
```

⟨Henrici-Brown-Output 74⟩ ≡

```
// File created by scm2cpp
// hb+2.type -> hb+2.type.cpp

#include <iostream.h>

template<class q, class i>
q & operator+(const q & r, const q & s) {
  i r1;
  i r2;
  i s1;
  i s2;
  i t1;
  i t2;
  i r2_H__BS_star;
  i s2_H__BS_star;
  i t1_H__BS_star;
  i t2_H__BS_star;
  i d;
  i e;
  q (*t) = new q;
  if ((r == 0)) {
    (*t) = s;
    return (*t);
  }
  if ((s == 0)) {
    (*t) = r;
    return (*t);
  }
  r1 = num(r);
  r2 = denom(r);
  s1 = num(s);
  s2 = denom(s);
  if (((r2 == 1) && (s2 == 1))) {
    (*t) = fract((r1 + s1), 1);
    return (*t);
  }
  if ((r2 == 1)) {
    (*t) = fract(((r1 * s2) + s1), s2);
    return (*t);
  }
  if ((s2 == 1)) {
    (*t) = fract(((s1 * r2) + r1), r2);
    return (*t);
  }
  d = gcd(r2, s2);
  if ((d == 1)) {
    (*t) = fract(((r1 * s2) + (r2 * s1)), (r2 * s2));
    return (*t);
  }
  if ((d != 1)) {
    r2_H__BS_star = (r2 / d);
    s2_H__BS_star = (s2 / d);
    t1 = ((r1 * s2_H__BS_star) + (s1 * r2_H__BS_star));
    t2 = (r2 * s2_H__BS_star);
    if ((t1 == 0)) {
      (*t) = 0;
      return (*t);
    }
    if ((t1 != 0)) {
      e = gcd(t1, d);
      t1_H__BS_star = (t1 / e);
      t2_H__BS_star = (t2 / e);
      (*t) = fract(t1_H__BS_star, t2_H__BS_star);
    }
  }
  return (*t);
}
// --- EOF by scm2cpp ---
◇
```

Macro never referenced.

During translation the following was displayed on the terminal

⟨Henrici-Brown display 76a⟩ ≡

```
Parsing SuchThat input hb+2.sth -> hb+2.scm
Attribute association hb+2.scm -> hb+2.staa hb+2.pp1
Typecheck hb+2.staa -> hb+2.type hb+2.pp2 hb+2.imp sttype.err sttype.log
sttype.err contains 81 warnings and 0 errors.
Generating C++ output hb+2.type -> hb+2.cpp
scm2cpp v1.0
Errors while parsing: 0
Errors during code generation: 0
compilation took .36 seconds
◇
```

Macro never referenced.

B.3 Demonstration of the Instantiation

The following example has been included in section 2.2 in full length and with detailed explanations. We list it here for completeness of this section.

⟨Example for the instantiation process 76b⟩ ≡

```
Global: Structure FiniteSequence over Set is abstract.

Algorithm: BubbleSort(a ; b)
        Let O be Set with O is ordered;
        Let FSQ(O) be FiniteSequence over O;
        Let F be FSQ(O).
Input: $a \in$ F.
Output: $b \in$ F. ||

Algorithm: main()
Local: $A \in$ Array(Integer).
(1) BubbleSort(A ; A) ||
◇
```

Macro never referenced.

The following messages mean that our function call has been accepted:

⟨example output 76c⟩ ≡

```
*** report 2 #80 ***
(prototypes found for subexpressions are  #42)

*** report 2 #81 ***
((bubblesort a) :
 ((((bubblesort ((a @f)) out ((b @f)))
out
 (((b (app_par array integer))))
bindings
 ((((be f (over finitesequence @o)) (app_par array integer))
---
))))))
◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

C The Implications Used

Since STGENTZ has not been connected to the type checker yet, we are forced to state all implications explicitly. Whenever no prototype can be found (which is an error condition) for an expression, the warnings preceding the error messages contain the implications that could not be looked up:

⟨example output 77a⟩ ≡

```
warning: implication not found
(field ring . #t)
◇
```

Macro defined by scraps 7b, 8ac, 9a, 10abc, 11, 76c, 77a.
Macro never referenced.

They can be cut-and-pasted into the following file, stating whether they are #t or #f.

"cache.imp" 77b ≡

```
(
 (Integer Set . #t)
 (Integer Ring . #t)
 (Integer Field . #f)
 (Integer IntegralDomain . #t)
 (Integer Natural . #f)
 (Integer Real . #t)
 (Integer gcddomain . #t)
 (integer (of fractionfield gcddomain) . #f)
 (integer (with (of fractionfield gcddomain) normalized) . #f)
 (integer euclideanring . #t)

 (Field   Ring . #t)

 (Ring Integer . #f)
 (Ring Field . #f)
 (Ring Set . #t)

 (Natural Integer . #t)
(Natural Set . #t)
(Natural gcddomain . #t)
(natural (of fractionfield gcddomain) . #f)
(natural (with (of fractionfield gcddomain) normalized) . #f)
(natural euclideanring . #t)

(Real Set . #t)
(Real (with Set has_plus_op) . #t)
(Real Ring . #t)
(Real Field . #t)

(Rational Ring . #t)
(Rational Field . #t)

(FractionField Ring . #t)
(FractionField Field . #t)
(FractionField Set . #t)

(IntegralDomain Set . #t)

; for the hb example
((of FractionField gcdDomain) Set . #t)
(gcdDomain Set . #t)

(gcddomain (of fractionfield gcddomain) . #f)
((with gcddomain normalized)
 (with (of fractionfield gcddomain) normalized) . #f)

(euclideanring gcddomain . #t)
(euclideanring set . #t)
(euclideanring integral_domain . #t))
```
◇

D Index

D.1 Files

D.2 stweb Macros

D.3 Identifiers