The FTCRL Reference Guide
(for Version 1.0)

Automating Test Case Refinement

Maximiliano Cristiá cristia@cifasis-conicet.gov.ar

CIFASIS
Bv. 27 de febrero 210 bis
(2000) Rosario
Argentina

Release date: August 2014

Contents

1 Introduction

In this document we present a language called Fastest Test Case Refinement Language (FTCRL). FTCRL is a general method for translating abstract test cases produced by Fastest into a set of programs of some programming language. The method receives a user-defined refinement rule for a given Z operation, a list of test cases for that operation and the name of a programming language, and applies the refinement rule to the list of test cases outputting a list of programs, each of which:

1. Sets the initial state of the SUT as specified by the test case.
2. Sets the input parameters waited by the SUT as specified by the test case.
3. Calls the SUT.

Each of these programs is a *concrete test case*—or just a test case when it is clear from context. Refinement rules are written in FTCRL which is a declarative language, in principle, independent of any programming language.

All the remaining test activities—i.e. compiling test cases, executing them, capturing their output, etc.—are beyond the scope of this document.

1.1 Principles for a Test Case Refinement Language

Test case refinement is far more simple than a general refinement calculus [Mor94, BW98] because only constant values of state and input variables are refined. The problem is further simplified since we assume that users will define each refinement rule. Therefore, the key issue is to define a sufficiently general TCRL such that:

1. It is no more complex than Z or a programming language.
2. Its refinement rules are independent of any programming language—users can refine the same test cases to different programming languages by applying the same refinement rule.
3. It is possible to define how any type and variable of the model must be refined into any data structure—if it is impossible to define the refinement of an element appearing in the specification, it should be due to an error in the corresponding implementation.
4. Once the refinement rule has be defined, the refinement of the list of test cases is completely automatic.
5. Test cases do not use the SUT to get it ready to be tested—because there is no guarantee that the SUT works according to its specification.

From these points it can be deduced that if the implementation of some variable in the VIS of some Z operation is changed, then it will be necessary to change the corresponding refinement rule accordingly[1]. For instance, if a function was implemented as an array but it is changed to a linked list, then the refinement rule has to be updated.

We cannot prove that FTCRL meets all those points—except number 4—but we are confident that if it does not, it can be extended to do so.

[1]Obviously, the same is true if the model is changed, but from the MBT perspective the model is assumed to be correct.

1.2 FTCRL's Philosophy

FTCRL is an interpreted language whose programs are the refinement rules. The interpreter receives the target programming language as a parameter. Therefore, strictly speaking, FTCRL semantics depends on the target programming language because the object code that is generated depends on that parameter.

FTCRL programs receive a list of abstract test cases generated by Fastest as their sole input, and produce a list, of the same length, of concrete test cases in the target programming language.

In principle, Z specifications can be implemented in many different ways. Therefore, an operation can be the specification for several units. For instance, an operation may specify what should be listed when the user chooses some menu option, but this listing is implemented in several output formats (PDF, HTML, text, etc.). Then, for a given operation there might be several units implementing it in different ways in the same system. When talking about these units we will say *unit under test* (UUT).

The intention behind FTCRL is that users should define one refinement rule for each (*operation*, *UUT*) pair. Sometimes, these refinement rules will barely differ from each other, sometimes not. Each refinement rule is essentially a description specifying how each VIS variable must be refined into some input variable(s) of the implementation. In this way, each concrete test case will be able to set the initial state for the test, to set the values for the input parameters waited by the UUT and to call the UUT. Among the UUT input variables we include files, tables in a data base, input parameters declared in the UUT's signature, global variables accessed by the UUT, and so on. In general, there is not a one-to-one correspondence between VIS variables and UUT variables. A refinement rule must ensure that every UUT variable is initialized before the UUT is called.

From the preceding paragraph, it can be deduced that Fastest's users need to know implementation details of UUT's when writing refinement rules. More precisely, we assume testers have access to the source code of the SUT and, particularly, that they can learn the meaning of the variables and data structures used in the SUT.

1.3 A FTCRL Example

In this section we introduce a typical refinement rule; we will explain and analyse it in the following sections. Consider the Z specification shown in Figure 1. Let's say that Fastest has generated the following abstract test cases:

$$
\begin{aligned}
&NewClient_1^{ATC} == [NewClient_2^{SP} \mid balances = \varnothing \land name? = name0 \land \\
&\quad n? = an0 \land u? = uid0 \land clients = \varnothing \land owners = \varnothing] \\
&NewClient_2^{ATC} == [NewClient_4^{SP} \mid u? = uid0 \land name? = name0 \land \\
&\quad n? = an0 \land balances = \{(an1, 20)\} \land \\
&\quad clients = \{(uid1, name0)\} \land owners = \{(uid1, an1)\}] \\
&NewClient_3^{ATC} == [NewClient_2^{DNF} \mid balances = \varnothing \land name? = name0 \land \\
&\quad n? = an0 \land u? = uid0 \land clients = \{(uid0, name0)\} \land owners = \varnothing] \\
&NewClient_4^{ATC} == [NewClient_3^{DNF} \mid n? = an0 \land name? = name0 \land \\
&\quad balances = \{(an0, 0)\} \land u? = uid0 \land clients = \varnothing \land owners = \varnothing]
\end{aligned}
$$

Assume this banking system is implemented in C[2]. Let's say that elements of *AN* and *NAME* are implemented as character strings, elements of *UID* are integer numbers and those of *BALANCE* are floats. Say *clients* is implemented as a linked list, `c`, declared as:

```
struct cdata {int uid; char *name; struct cdata *n;} *c;
```

[2]We also assume the reader is familiar with the C programming language [KR88].

$[AN, UID, NAME]$
$BALANCE == \mathbb{N}$

Bank
$clients : UID \pfun NAME$
$balances : AN \pfun BALANCE$
$owners : UID \rel AN$

NewClientOk
$\Delta Bank$
$u? : UID$
$name? : NAME;\ n? : AN$

$u? \notin \mathrm{dom}\ clients$
$n? \notin \mathrm{dom}\ balances$
$clients' = clients \cup \{u? \mapsto name?\}$
$balances' = balances \cup \{n? \mapsto 0\}$
$owners' = owners \cup \{u? \mapsto n?\}$

$ClientAlreadyExists == [\Xi Bank;\ u? : UID \mid u? \in \mathrm{dom}\ clients]$
$AccountAlreadyExists == [\Xi Bank;\ n? : AN \mid n? \in \mathrm{dom}\ balances]$
$NewClient == NewClientOk \lor ClientAlreadyExists \lor AccountAlreadyExists$

Figure 1: A partial Z specification of the savings accounts of a banking system.

balances is implemented as an array, `b`, declared as:

```
struct bdata {char* num; float bal;} b[100];
```

and there is an integer variable, `l`, pointing to the last used component of `b`. *owners* is implemented as doubled-linked list, `o`, declared as:

```
struct odata {int *puid; char *pn; struct odata *n,*p;} *o;
```

where `puid` should point to the `uid` member of a node in `c`; `pn` should point to the `num` member of a `b` component; and `n` and `p` are pointers to the next and previous nodes in the list, respectively. Say that `c`, `b`, `l` and `o` are global variables. Finally, let's assume that *NewClient* is implemented by a C function with the following signature:

```
int newClient(int u, char *name, char *n)
```

Then, the refinement rule is shown in Figure 2 and the concrete test case generated by applying it to $NewClient_2^{TCASE}$ is shown in Figure 3. The reader will need to look at these figures while we introduce FTCRL below.

1.3.1 The Basic Structure of a Refinement Rule.

The first line in a refinement rule is its name, which can be used in future references as we will show. Refinement rules have four sections that must be written in strict order: `@PREAMBLE`, `@LAWS`, `@UUT` and `@EPILOGUE`. The preamble and the epilogue must contain legal source code—in one of the programming languages supported by Fastest. The preamble is blindly copied at the beginning of a test case, and the epilogue right at the end. The preamble should contain all the code that it is necessary to compile and run the test case—for instance, UUT's definition, type declarations, sentences to import external resources, header files, etc. It is assumed that the preamble contains the definition of a function named `init` without parameters. This function returns 0 or 1 and it is called right before each test case is executed. The epilogue should contain code to perform clean-up once the test has been run—for instance, deleting a file. The `@UUT` section contains only one line of code to call the UUT. The parameters of the subroutine call must be identifiers not used in the

```
@RRULE bank
@PREAMBLE
#include <bank.h>
int init() {;}
@LAWS
l01:u?       ==> u
l02:name?    ==> name
l03:n?       ==> n
l04:clients  ==> c AS LIST[SLL,n] WITH[s01;s02]
l05:balances ==> b AS ARRAY WITH[s03;s04];
                 balances.@# ==> l
l06:owners   ==> o AS LIST[DLL,n,p] WITH[s05;s06]
l07:s01 == clients.@DOM ==> cdata.uid
l08:s02 == clients.@RAN ==> cdata.name
l09:s03 == balances.@DOM ==> bdata.num
l10:s04 == balances.@RAN ==> bdata.bal
l11:s05 == owners.@DOM ==> odata.puid AS REF[c.uid]
l12:s06 == owners.@RAN ==> odata.pn AS REF[b.num]
@UUT newClient(u,name,n)
```

Figure 2: Refinement rule for *NewClient*.

```
#include <bank.h>
int main() {
if (!init()) {printf("Initialization error\n"); return 1;}
int u = 345;
char *name = "name0", *n = "an0";
struct cdata cdata0 = {87,"name0",NULL};
struct bdata bdata0 = {"an1",20};
struct odata odata0 = {0,0,NULL,NULL};
c = &cdata0;
b[0] = bdata0;
l = 1;
odata0.puid = &cdata0.uid;
odata0.pn = bdata0.num;
o = &odata0;
newClient(u,name,n);
return 0;
}
```

Figure 3: Concrete test case for $NewClient_2^{ATC}$ generated by `bank` of Figure 2.

UUT. The value returned by the UUT is not considered since it does not affect refinement, but other steps of the MBT process.

The name of a refinement rule can be used in other refinement rules as follows, with the obvious meaning:

```
@RRULE otherBankingRefRule
@PREAMBLE bank.@PREAMBLE
@LAWS
bank.l04
.....
@UUT deposit(....)
@EPILOGUE system("rm *");
```

Note that this mechanism allows users to use the same `@LAW` with different preambles and epilogues, thus making it possible to refine the same abstract test cases to different programming languages, since all the FTCRL code that depends on the programming language of the SUT is confined to these two sections.

1.3.2 Refinement Laws.

The `@LAWS` section is a list of *refinement laws* (or just laws) and *synonymous*. Each line in this section is preceded by an identifier followed by a colon character. This identifier can be used to include the law in other refinement rules, as shown in the previous section. A synonymous is of the form:

```
ident:ident == FTCRL code
```

where the second identifier can be used in a law as a synonymous for the FTCRL code at the right of the `==` token. For instance, by defining the following synonymous:

```
l13:s06 == balances.@# ==> 1
```

we could have written `l05` as:

```
l05:balances ==> b AS ARRAY WITH[s03;s04;s06]
```

A refinement rule whose `@LAW` section contains only synonymous produces no concrete test cases. Synonymous are preprocessed by the interpreter.

The basic form of a law is:

```
ident:list_of_spec_var ==> refinement
```

where `list_of_spec_var` is a list of one or more variables declared in the specification, and `refinement` specifies how the specification variables must be refined. The token `==>` can be read as 'refines to'. The most simple law is, for instance, `l01` in Figure 2. For each abstract test case, this law makes the interpreter to declare a local variable named `u` of type `int`—the type is deduced by parsing the definition of `newClient`—and to assign it the value of $u?$ in the abstract test case. Constant values at the specification level, such as $uid0$, are traduced to the implementation type by applying an arbitrary bijection whenever necessary.

Law `l04` specifies that *clients* is implemented as a list. The first parameter between square brackets indicates that `c` is a a simply-linked list and the second one is the name of the variable pointing to the next node in the list. It is necessary to include this information in the law because it is impossible to automatically deduce that `c` is a list, solely from its declaration. Since *clients* is not

a sequence, we must say how each component is refined. In this case, as `s03` and `s04` show, elements in the domain go to `uid` and elements in the range to `name`, both of `cdata`. Therefore, the interpreter creates a new variable of type `cdata` for each pair in *clients* and initializes them with the constant values of each pair. The value of the `n` member of each of those new variables is set to point to the address of one of them—since *clients* is a function, there is no order between its pairs, and so any order in `c` should be correct.

Note how a specification variable is refined to more than one implementation variable in `l05`. The expression `balances.@#` means the cardinality of *balances*. Had it be necessary to make `l` to point to the first free component in `b`, then we would have written: `balances.@# + 1`—in general, any constant expression is valid.

Regarding `l06`, `DLL` stands for double-linked list—we also have defined `CLL` for circular-linked list and `DCLL` for circular double-linked list— and the other two parameters are the members pointing to the next and previous nodes, respectively. If an implementation variable is intended to hold a reference (or a pointer) to some data in some other data structure, the `REF` directive must be used. It is possible to generate source code according to this specification because every element of a dynamic data structure is first saved in a new static variable whose name, memory address and value can be freely used by the interpreter. In turn, creating new variables for each element in a test case is possible because abstract test cases are defined by finite sets of values.

Observe that all the typing information can be deduced by parsing both the LaTeX markup of the Z specification and the source code of the SUT.

2 FTCRL Syntax, Type Restrictions and Style

In this section the FTCRL syntax is described by giving the intended usage of each legal construction. Reading the semantics sections will be necessary from time to time. Remember that:

- Strings in `typewriter type` are terminal strings.
- *eol* means "end of line".

All the examples given in this section are in the C programming language.

2.1 Refinement Rules

⟨*refinementRule*⟩ ::= `@RRULE` ⟨*name*⟩ *eol*
[⟨*preamble*⟩ *eol*]
[⟨*laws*⟩ *eol*]
[⟨*plcode*⟩ *eol*]
[⟨*uut*⟩ *eol*]
[⟨*epilogue*⟩ *eol*]

Each refinement rule has a name that can be used in other refinement rules to refer to it, as shown in the following sections. Each section must start in a new line. All sections are optional. When executing a refinement rule, code is generated only if it has a ⟨*laws*⟩ and ⟨*uut*⟩ sections. The other forms of a refinement rule can be used to define elements that are later reused in other refinement rules.

2.2 The Preamble

⟨*preamble*⟩ ::= `@PREAMBLE` *eol*
⟨*PLCode*⟩ | ⟨*name*⟩`.@PREAMBLE` *eol*
{⟨*PLCode*⟩ | ⟨*name*⟩`.@PREAMBLE` *eol*}

The preamble must contain legal source code in the programming language used to implement the UUT. The preamble is blindly copied at the beginning of a test case. The preamble should contain all the code that it is necessary to compile and run the test case—for instance, UUT's definition, type declarations, sentences to import external resources, header files, etc. Fastest assumes the code is correct and compiles. In particular, it must contain the declaration of every implementation variable mentioned in the `@LAWS` section, except perhaps the parameters appearing in the `@UUT` section, and if necessary, the definition of their types. This can be achieved by explicitly writing this code or by including some header files or similar resources.

It is assumed that the preamble contains the definition of a function with the following signature:

```
int init()
```

In other words, this function receives no parameters and returns 0, meaning that there was some error during the execution, or 1, meaning that it executed successfully. This function is called right before each test case is executed. Since function definition and function call is a language-dependent construction, the reader must refer to the sections describing the semantics of TCRL for each programming language to learn more details.

Preambles can be reused. The preamble of refinement rule `A` can be included anywhere in the preamble of refinement rule `B` by writing `A.@PREAMBLE`. The preamble of a given refinement rule can contain the preamble of any other refinement rule. In any case, sentences of the form `A.@PREAMBLE` are replaced by the respective preambles, before the interpreter attempts to generate test cases.

2.3 The @LAWS Section

⟨*laws*⟩ ::= `@LAWS` *eol*
⟨*law*⟩ *eol* | ⟨*lawReference*⟩ *eol* | ⟨*name*⟩`.@LAWS` *eol*
{⟨*law*⟩ *eol* | ⟨*lawReference*⟩ *eol* | ⟨*name*⟩`.@LAWS` *eol*}

⟨*law*⟩ ::= [⟨*name*⟩`:`]⟨*refinementLaw*⟩

⟨*lawReference*⟩ ::=
⟨*name*⟩`.`⟨*lawName*⟩[`[`⟨*varSubst*⟩{`,`⟨*varSubst*⟩}`]`[⟨*varSubst*⟩{`,`⟨*varSubst*⟩}`]`]

⟨*varSubst*⟩ ::= ⟨*name*⟩`<-`⟨*name*⟩

The `@LAWS` section is a list of *refinement laws* (or just laws) or sentences including the laws of other refinement rules. Each law takes one or more lines of text; each inclusion sentence takes exactly one line. Each law may be preceded by an identifier followed by a colon character. This identifier can be used to include the law in other refinement rules' `@LAWS` section by using the dot notation, as shown below.

```
@RRULE A
@PREAMBLE
B.@PREAMBLE
@LAWS
B.104
.....
C.new
.....
D.@LAW
.....
@UUT f(....)
@EPILOGUE system("rm *");
```

It is also possible to include the entire `@LAWS` section of others refinement rules. This has the effect of copying all the lines in that sections in the current refinement rule. Individual and grouped references are replaced by the corresponding definitions before the code generation phase starts.

The order in which laws and references are placed inside the `@LAWS` section has no semantic effect.

A refinement rule whose `@LAW` section is empty produces no concrete test cases.

A reference to a law can include two substitution lists. For example:

```
C.oldLaw[a <- kk][num <- m]
```

Both lists may be empty but if one is not empty the other one must be present (with an empty pair of square brackets).

The first substitution list correspond to the left hand side of a refinement law; the second one, to the right hand side of the same refinement law. The names at the left hand side of each `<-` token of the first substitution list must be names appearing at the left hand side of the referenced refinement law. The names at the left hand side of each `<-` token of the second substitution list must be names appearing at the right hand side of the referenced refinement law. For example, in the code shown above `a` must be a variable name appearing at the left hand side of `oldLaw`; and `num` must be a variable name appearing at the right hand side of `oldLaw`. The names appearing at the right hand side of each `<-` token can be any names.

The effect of the substitutions is to substitute the names at the left hand side of each `<-` token by those at the right hand side. So if in refinement rule `C` we have:

```
oldLaw: a ==> num
```

then

```
C.oldLaw[a <- kk][num <- m]
```

will mean

```
kk ==> m
```

In such a simple example the utility of the substitution mechanism may be not obvious, but in more complex laws it certainly will be helpful.

2.4 Programming Language Code

$\langle plcode \rangle$ ::= `@PLCODE[`$\langle varSubst \rangle${, $\langle varSubst \rangle$}`]` *eol*
$\langle PLCode \rangle$ *eol*

This section is optional and should be used only in very complex refinement rules; it will be useful only to refine to very complex or unsupported data structures. The contents of this section must be legal source code in some programming language and it will be blindly copied right before the call to the UUT.

If some specification variable must be refined into some data structure for which FTCRL falls short, then testers have the opportunity to solve the problem in three steps by using this section:

1. Create a file including the UUT and declaring some auxiliary variables whose types are such that code can be written to complete the refinement to the unsupported or complex data structures.

2. Write refinement laws to refine the conflicting specification variables to the auxiliary variables declared above.

3. Write a program (in the `@PLCODE` section) to complete the refinement by conveniently binding the auxiliary variables to the real variables.

Let's assume for a moment that record-like data structures were not supported by FTCRL. Say $r : NAME \times \mathbb{N}$ is a specification variable, where *NAME* is a given type. Say elements of *NAME* must be refined as character strings and elements of $\mathbb{N}$ must be refined as `int`. Assume r must be refined to a C structure as follows:

```
struct x {int n; char *name;} rr;
```

If the UUT is declared in `uut.h`, then the first step is to create another header file called, say, `myuut.h`:

```
#include <uut.h>

int auxn;
char *auxname;
```

The second step is to write the following `@LAWS` section:

```
@LAWS
r ==> r.1 ==> auxname; r.2 ==> auxn;
```

And finally, write the following `@PLCODE` section (in this case it should be C code):

```
@PLCODE[rr,auxname,auxn]
rr.n = auxn;
rr.name = auxname;
```

The identifiers between square brackets must be all the indentifiers used in the right hand side of the refinement law (`auxname` and `auxn` in the example) and those to which the final refinement must be done (`rr` in the example). These parameters to the `@PLCODE` section can then be substituted if the section is included in other refinement rules, as has been shown for refinement laws.

2.5 The Unit Under Test

⟨*uut*⟩ ::= `@UUT` ⟨*iName*⟩`(`[⟨*iName*⟩{`,`⟨*iName*⟩}]`)` [`MODULE` ⟨*iName*⟩] *eol*

This section contains only one line of code to call the UUT. The parameters of the subroutine call must be identifiers not used in the UUT. The function call will be rewritten according to the syntax of the target programming language during the code generation phase. This call is placed right after all the global variables, input parameters and external resources, used in the UUT, have been initialized.

The `MODULE` directive is useful in some programming languages because functions may belong to some entities such as modules or classes. It is not intended to be used to indicate the file where the UUT is defined since this information is assumed to be available in the preamble section. In some languages this directive is discarded.

The value returned by the UUT is not considered since it does not affect refinement, but other steps of the MBT process.

2.6 The Epilogue

⟨*epilogue*⟩ ::= `@EPILOGUE` *eol*
⟨*PLCode*⟩ | ⟨*name*⟩`.@EPILOGUE` *eol*
{⟨*PLCode*⟩ | ⟨*name*⟩`.@EPILOGUE` *eol*}

The epilogue must contain legal source code in the programming language used to implement the UUT. The epilogue should contain code to perform clean-up once the test has been run—for instance, deleting a file. This section is blindly copied right after the call to the UUT in each test case.

2.7 Refinement Laws

⟨*refinementLaw*⟩ ::= ⟨*sName*⟩ {`,`⟨*sName*⟩}
`==>` ⟨*refinementSentence*⟩{`;`⟨*refinementSentence*⟩}

⟨*refinementSentence*⟩ ::= ⟨*sName*⟩ {`,`⟨*sName*⟩} `==>` ⟨*refinementSentence*⟩
| ⟨*refinement*⟩
| ⟨*lawReference*⟩

A refinement law specifies how one or more specification variables (⟨*sName*⟩) are implemented. Specification variables are at the left hand side of the `==>` token, and the specification of the refinement at the right hand side. The symbol `==>` reads as "refines to" or "implemented by". The implementation of specification variables is described by indicating which program variables (or implementation variables) are related to them. Program variables are, for now, hidden in the ⟨*refinement*⟩ non-terminal.

In this context, a refinement specification consists essentially in initializing implementation variables. Each refinement law must completely initialize all of its implementation variables (this includes members of record-type variables). Therefore, the left hand side of a refinement law must list the minimum set of specification variables to be able to verify the previous condition for the implementation variables at the right hand side. A specification variable can appear in one and only one refinement law. The left hand side of a refinement law must list more than one specification variable, only when their implementations are related to each other through one or more implementation variables. When one or more specification variables are refined by n implementation variables, then the refinement law must consist of n refinement sentences separated by a semicolon. Each refinement sentence must talk about only one implementation variable. In FTCRL the semicolon is commutative, so you can write refinement sentences in any order. The ⟨*sName*⟩'s appearing in a ⟨*refinementSentence*⟩ must be a proper subset of the ⟨*sName*⟩'s listed in a previous refinement sentence. Every refinement sentence must end in a ⟨*refinement*⟩ non-terminal and all the ⟨*sName*⟩'s listed before it must be used in it. That is, if a refinement sentence is of the form:

$x_1, \ldots, x_n$ `==>` ⟨*refinement*⟩

then ⟨*refinement*⟩ must depend on all $x_1, \ldots, x_n$.

The ⟨*refinement*⟩ non-terminal is as follows:

⟨*refinement*⟩ ::= [⟨*sExprRefinement*⟩ `==>`]⟨*iExprRefinement*⟩

where ⟨*sExprRefinement*⟩ is an expression depending on specification variables that will be explained in detail in Section 2.8, and ⟨*iExprRefinement*⟩ is:

⟨*iExprRefinement*⟩ ::= ⟨*iName*⟩[`AS` ⟨*asRefinement*⟩] | `@UINPUT` | `@SYSDATE`

where ⟨*iName*⟩ is an implementation variable, a regular expression matching some implementation variables or a few more things that will be shown shortly; `@UINPUT` represents user input and it is explained in Section 2.10; `@SYSDATE` represents the system date and it is explained in Section 2.11. An implementation variable can appear in one and only one refinement law unless it is the parameter of one or more `REF` clauses (see Section 2.9.5), but in this case this implementation variable can appear only once outside a `REF` clause.

Let's see some examples to illustrate the previous grammar. Consider the following excerpt from some specification:

$[NAME]$
$AddPerson == [first?, last? : NAME \ldots \mid \ldots]$

Assume the implementation stores the first and last name of persons in a single character string variable, `name`. Then, the refinement law is as follows:

```
person:first?, last? ==> last? ++ ", " ++ first? ==> name
```

where `last? ++ ", " ++ first?` is a ⟨*sExprRefinement*⟩.

However, if there are two implementation variables, `fname` and `lname`, the following refinement law is wrong:

```
person:first?, last? ==> last?  ==> lname;
                         first? ==> fname;
```

since *first*? and *last*? are not related to each other through one or more implementation variables. The right refinement law would be:

```
last :last?  ==> lname;
first:first? ==> fname;
```

And if the specification would have been:

$[NAME]$
$AddPerson == [name? : NAME \ldots \mid \ldots]$

with the intention to use *name*? as the first and last name, then the refinement law could be:

```
name:name?  ==> name? ==> lname; name? ==> fname;
```

or

```
name :name?  ==> name? ++ ", " ++ name? ==> name
```

depending on the implementation.

As a final example, say *clients* is a Z variable whose type is $UID \pfun NAME$ where *UID* and *NAME* are basic types. Assume this variable is implemented as two separate arrays of the same length and two integers pointing to the last used position of each array; each array stores the elements in the domain and range, respectively. Hence, the refinement law would be:

```
104:clients ==>  clients.@DOM ==> uids AS ARRAY;
                 balances.@CARD  ==> last_u;
                 clients.@RAN ==> names AS ARRAY;
                 balances.@CARD  ==> last_n;
```

where `clients.@DOM`, `balances.@CARD`, `clients.@RAN` and `balances.@CARD` are all $\langle sExprRefinement \rangle$.

Observe, as the examples show, that a specification variable can be used to implement several implementation variables and can be implemented by several of them.

Refinement specifications can have one of three forms:

- An implementation variable. This form is used for variables of simple types such as integers, characters or strings.
- An implementation variable followed by an `AS` directive. This form is used when it is not clear how the implementation variable must be used. See next subsection.
- A refinement expression. This form is used when not the specification variable but an expression involving it must be refined to some implementation variable. See section 2.8.

2.7.1 The AS and WITH Directives

⟨*asRefinement*⟩ ::= ⟨*dataStruct*⟩[WITH[⟨*refinement*⟩{;⟨*refinement*⟩}]]

⟨*dataStruct*⟩ ::= ARRAY
| RECORD
| MAPPING
| ⟨*list*⟩ | ⟨*reference*⟩ | ⟨*enumeration*⟩ | ⟨*table*⟩ | ⟨*file*⟩

Sometimes it is necessary to indicate the intended usage of an implementation variable because it is impossible to deduce it automatically. For instance, in the C programming language the following declaration:

```
struct cdata {int uid; char *name; struct cdata *n;} *c;
```

says that `c` is a pointer to an object of type `cdata`. However, `c` can be the first node of a simply-linked list. The fact that the implementation uses `c` as a linked list cannot be deduced from its declaration. Therefore, the refinement sentence must state that fact. The role of an implementation variable is stated with the `AS` directive.

At the right of the `AS` token the user can say that an implementation variable is used as an array, a record, a list, a pointer or reference to another entity, an enumeration, a table in a database, a file in the file system or a mapping. See the details in Section 2.9.

If an implementation variable is used as indicated by its type, then there is no need to explicit its usage. This is usually the case for variables of basic types such as integers, strings, etc. For example, say that *UID*, *AN* and *NAME* are basic types of some Z specification. Assume that *AN* and *NAME* are implemented as character strings and elements of *UID* are integer numbers. Then, the refinement laws for $u? : UID$, $name? : NAME$ and $n? : AN$ are as follows, assuming their implementation counterparts have the same names:

```
l01:u?       ==> u
l02:name?    ==> name
l03:n?       ==> n
```

The optional directive `WITH` must be used when different parts of the specification variable must be refined into different parts of one implementation variable. Consider the following example. Say *clients* is a Z variable whose type is $UID \pfun NAME$ where *UID* and *NAME* are basic types. Assume it is implemented as a simply-linked list, `c` declared as above, and where the first component of each ordered pair in *clients* should be stored in `uid`, the second component in `name`, and `n` is a pointer to the next node in the list. Then, the refinement law is as follows:

```
l04:clients  ==> c AS LIST[SLL,n] WITH[clients.@DOM ==> cdata.uid;
                                       clients.@RAN ==> cdata.name]
```

Note that inside a `WITH` directive it is possible to access the "conceptual type" of `c`. In effect, at a conceptual level `c` is a list of type `cdata`, i.e. it stores elements of that type. This is also possible for other data structures such as arrays, mappings, files, etc.

2.8 Refinement Expressions ($\langle sExprRefinement \rangle$)

$\langle sExprRefinement \rangle ::= \langle sName \rangle$
$\quad | \langle zExprSet \rangle$
$\quad | \langle zExprNum \rangle$
$\quad | \langle zExprString \rangle$
$\quad | \langle zExprSeq \rangle$
$\quad | \langle funAppExpr \rangle$

$\langle zExprSet \rangle ::= \langle sName \rangle [\texttt{.} \langle dotSetOper \rangle]$
$\quad | \langle setExtension \rangle$
$\quad | \langle zExprSet \rangle$ `@CUP@` $\langle zExprSet \rangle \mid \ldots$

$\langle zExprNum \rangle ::= \langle sName \rangle [\texttt{.@\#}]$
$\quad | \langle number \rangle$
$\quad |$ `@AUTOFILL`
$\quad | \langle zExprNum \rangle$ `div` $\langle zExprNum \rangle$
$\quad | \langle zExprNum \rangle$ `/` $\langle zExprNum \rangle$
$\quad | \langle zExprNum \rangle$ `div` $\langle zExprNum \rangle$
$\quad | \langle zExprNum \rangle$ `mod` $\langle zExprNum \rangle$
$\quad | \langle zExprNum \rangle$ `+` $\langle zExprNum \rangle$
$\quad | \ldots$

$\langle zExprString \rangle ::= \langle string \rangle$
$\quad | \langle number \rangle$
$\quad |$ `@AUTOFILL`
$\quad | \langle sName \rangle [\texttt{.}(\langle dotSetOper \rangle \mid$ `#` $\mid$ `@STR`$)]$
$\quad | \langle zExprString \rangle$ `++` $\langle zExprString \rangle$

$\langle zExprSeq \rangle ::= \ldots$

$\langle funAppExpr \rangle ::= \langle iIdent \rangle$`(`$[\langle refinement \rangle \{$`,`$\langle refinement \rangle\}]$`)`

$\langle dotSetOper \rangle ::=$
$\quad$ `@(DOM | RAN | ELEM)` $\mid$ `@`$\langle digit \rangle \mid \langle digit \rangle \mid \langle sName \rangle \mid \langle dotSetOper \rangle \texttt{.} \langle dotSetOper \rangle$

If one of these expressions involves more than one set, then all of them must depend on the same specification variable. For example, if $A, b : \mathbb{P}\, X$ are two specification variables, then the following refinement sentence is invalid:

```
A, B ==> A ++ B ==> ...
```

because the expression `A ++ B` involves two sets that depend on different specifications variables (A and B). On the other hand, if $f : X \pfun Y$ is another specification variable then the following refinement sentences are correct:

```
A, B ==> A @CUP@ B ==> ...
f ==> f.@DOM ++ ":" ++ f.@RAN ==> ...
A, B, f ==> A @CUP@ B @CUP@ f.@DOM ==> ...
A, B ==> A ++ B.@STR ==> ...
```

because the first and third refinement expressions depend on only one set ($A \cup B$ and $A \cup B \cup \mathrm{dom}\, f$); the second one depends on two sets ($\mathrm{dom}\, f$ and $\mathrm{ran}\, f$) but they depend on the same specification variable (f); and the fourth depends also on only one set because `B.@STR` is not a set although B is a set (see below).

The ⟨*sExprRefinement*⟩ expressions can be of one of the following kinds:

- Expressions involving set operators (⟨*zExprSet*⟩). These can be any expression depending on one or more of the specification variables being refined in the current refinement law. The expressions can be formed with specification variables, whose types are of the form $\mathbb{P}\, X$ for some type X, and the Z set operators plus:

  - *var*`.@DOM` where *var* is one of the specification variables of the current law. This expression equals the domain of *var*, therefore its type must be of the form $\mathbb{P}(X \times Y)$ for any types X and Y.
  - *var*`.@RAN` where *var* is one of the specification variables of the current law. This expression equals the range of *var*, therefore its type must be of the form $\mathbb{P}(X \times Y)$ for any types X and Y.
  - *var*`.@ELEM` where *var* is one of the specification variables of the current law. This expression is valid only when *var* is of type $\mathbb{P}(\mathbb{P}\, X)$ for any type X. In this case, is means any (and all but one by one) of the elements (i.e. sets) of *var*.

    Since FTCRL processes sets by considering each of their elements by simply referencing the set (that is, there is no need to mention an element of a set), when the variable is a set of sets the user has no way to say how to refine their elements. For instance, if we have a specification variable $logs : \mathbb{P}(\mathrm{seq}(\mathbb{N} \times EVENT))$ where $EVENT$ is an enumerated type, and we want to refine it into an array of lists of records, `logs`. Without the `ELEM` operator all we can say is:

    ```
    rlog: logs ==> logs AS ARRAY
    ```

    but it does not specify how each sequence of *logs* is refined. The right law is, then:

    ```
    rlog: logs ==> logs AS ARRAY
                       WITH[logs.@ELEM
                            ==> logs[] AS LIST WITH[logs.@ELEM.1 ==> log.t;
                                                   logs.@ELEM.2 ==> log.e]]
    ```

    where `logs.@ELEM` is each of the lists in *logs*.
  - *var*`.@`*digit* where *var* is one of the specification variables of the current law. This expression equals the $digit^{th}$ set in a cross product, therefore *var*'s type must be of the form $X_1 \times \cdots \times X_n$ for any types X_i with $i \in 1 \,..\, n$ and $digit \leq n$.
  - *var*`.`*digit* where *var* is one of the specification variables of the current law. This expression equals the $digit^{th}$ element of the set.

    This expression should be used when each element of the set must be refined into unrelated variables of the implementation. For example, if $C : \mathbb{F}\, X$ for some type X and you know that $\#C < 4$, then at implementation level each element may be a different parameter in a subroutine call.

    ```
    C ==> C.1 ==> v1;
          C.2 ==> v2;
          C.3 ==> v3
    @UUT mySub(v1, v2, v3)
    ```

If, in the current test case, the set does not contain enough elements as the sentence requires, then the interpreter must assign a null value to the implementation variables at the right hand side of the refinement sentences from which the set run out of elements. The order in which the set is processed is non-deterministic.

- *var*`.`*field* where *var* is one of the specification variables of the current law. This expression is valid only when *var* is of a schema type such that *field* is one of its free variables. In this case the expression has the same meaning than the equivalent Z expression.
- These operators can be nested in expressions like `f.@RAN.@RAN.1.a`, provided the types are correct.

- Expressions involving arithmetic operators. These can be any expression depending on one or more of the specification variables being refined in the current refinement law. The expressions can be formed with specification variables whose type is $\mathbb{Z}$ and the Z arithmetic operators plus:
  - *var*`.@CARD` where *var* is one of the specification variables of the current law. This expression equals the size of *var*, therefore its type must be of the form $\mathbb{P}\, X$ for any type X.
  - *number* is any constant integer number.
  - The `@AUTOFILL` operator which is replaced by some (not specified) default constant value. This clause is useful when specification variables are so abstract that they omit some data that is stored at implementation level.

    For instance, assume that some system stores the identification number, name, address and age of each client. Specifiers abstracted away these details retaining only the identification number and the name, in the state variable $clients : UID \pfun NAME$, where *UID* and *NAME* are basic types. However, at implementation level *clients* is implemented as a simply-linked list declared as:

    ```
    struct cdata {int uid, age; char *name; char *addr; struct cdata *n;} *c;
    ```

    where the meaning of all the members is clear except for `n` which is the pointer to the next node in the list. In this case the refinement law would be:

    ```
    c:clients ==> c AS LIST[SLL,n] WITH[clients.@DOM ==> cdata.uid;
                                       clients.@RAN ==> cdata.name;
                                       @AUTOFILL    ==> cdata.*]
    ```

    or equivalently:

    ```
    c:clients ==> c AS LIST[SLL,n] WITH[clients.@DOM ==> cdata.uid;
                                       clients.@RAN ==> cdata.name;
                                       @AUTOFILL    ==> cdata.age;
                                       @AUTOFILL    ==> cdata.addr]
    ```

    As it can be seen, the `@AUTOFILL` command is useful when a certain entity that at implementation level is represented by a large structured type, it is abstracted by a considerable more simple type at specification level. A typical example would be the i-node structure of a UNIX file system, where many of its members would not be represented in a Z model. However, for testing purposes it is necessary to initialize all of them.
  - The `/` operator. The result of an expression involving `/` will be a floating point number (float). If the result of an arithmetic operation must be an integer number, then `div` and `mod` must be used.

- So-called string expressions. These can be any expression depending on one or more of the specification variables being refined in the current refinement law. The expressions can be formed with the following elements:

  - Any constant string enclosed between double quotes.
  - Any constant integer number.
  - The `@AUTOFILL` operator which is replaced by some (not specified) default constant character string.
  - Any of the specifications variables being refined.
  - Any legal (well-typed) combination of `.@DOM`, `.@RAN`, *.digit*, *.field* and `.@CARD`, applied to one of the variables being refined.
  - *var*`.@STR` where *var* is one of the specification variables of the current law. This expression is valid only when *var* is of type $\mathbb{P}\, X$ or $\mathrm{seq}\, X$.
  - The concatenation of any of the previous elements, expressed with `++`.

  These expressions are converted to character strings as follows (remember that specification variables are replaced by their values when the list of abstract test cases is processed by the interpreter, then at the moment of doing this conversion all we have are constants):

  - Numbers are converted into their equivalent character strings.
  - If x is a constant value of a basic or enumerated type, then it is converted into its equivalent character string.
  - $x \mapsto y$ and (x, y) are converted into xy and this in turn is converted into a character string.
  - $\{x_1, \ldots, x_n\}$`.@STR` is converted into $x_1 \ldots x_n$ and in this turn is converted into a character string.
  - $\langle x_1, \ldots, x_n \rangle$`.@STR` is converted into $x_1 \ldots x_n$ and this in turn is converted into a character string.
  - $\{x_1, \ldots, x_n\}$ and $\langle x_1, \ldots, x_n \rangle$ are processed one element at a time as any other set. Thus, in this case n strings are produced.
  - These rules are applied recursively.

  Therefore, it is expected that the expression at the right hand side of the `==>` token is compatible with a character string.

  Observe that it is necessary to add a newline character at the end of the expression if it is expected that the next line be written in a new line of a file. Same considerations apply for other control characters.

- Expressions involving the application of an implementation function. These expressions are of the form:

  $$\langle \mathit{funAppExpr} \rangle ::= \langle \mathit{iIdent} \rangle \texttt{(}[\langle \mathit{refinement} \rangle \{\texttt{,} \langle \mathit{refinement} \rangle\}]\texttt{)}$$

  where $\langle \mathit{iIdent} \rangle$ is intended to be the name of a function at the implementation level, and the list of $\langle \mathit{refinement} \rangle$ equals in number the number of arguments of the function definition. It is assumed that this function is declared in the preamble and returns values of some implementation type. Also, the right hand side of each $\langle \mathit{refinement} \rangle$ non-terminal in the argument list,

must mention the corresponding formal parameter of the function definition (according to its position).

Informally, the semantic of this kind of expressions is as follows: first, all the arguments are refined according to the refinement rules given in the argument list; second, the function is called by passing it the result of each refinement rule in the argument list; third the result of this call is stored in some variable. For example, say $f : X \times Y \pfun V$ must be refined as a map which is implemented in the C programming language as an array, `c`, whose components are lists of characters. Assume, that each key of the map is the result of applying function `hash` to two strings, which returns an integer. Then the refinement rule can be:

```
map:f ==> hash(f.@DOM.1 ==> a, f.@DOM.2 ==> b)
          ==> c AS ARRAY WITH[f.@RAN ==> c[]]
```

This means that the integer returned by `hash` is used to set the position in the array where the range of f will be stored. For example, when an element of f, say $(x, y) \mapsto v$, is refined the following code is generated:

```
int i1;
char* s11, s12, s13;
s11 = "x";
s12 = "y";
s13 = "v";
i1 = hash(s11,s12);
c[i1] = &s13;
```

Another example of using a function call is when some data must be stored encrypted or compressed but this feature was omitted in the specification. Say that records of type `PlainData` must be encrypted and then stored in a list of encrypted data. At specification level only some abstraction of `PlainData` was considered; say it is a variable $data : \mathrm{seq}\, Data$. Encryption is performed by a function called `encrypt` which takes a value of type `PlainData` and an encryption key which is a string, and returns an instance of a record of type `EncData`. Then the refinement law is as follows:

```
enc: data ==> encrypt(data.@RAN ==> d, "encryption_key" ==> key)
          ==> ed AS LIST[SLL,n]
```

2.8.1 Some Examples Involving Refinement Expressions and the `WITH` Directive

Suppose it is necessary to refine the following variable:

$$f : X \pfun Y \pfun H \times W$$

where X, Y and W are given types and H is the schema $[a : A;\ b : B]$. Let's say that f must be refined as a simply-linked list, `fi`, whose nodes are:

```
struct xd {int x; yd* ys; struct xd* n;}
```

where `ys` is intended to be a list of nodes whose type is:

```
struct yd {float y; hd* h; char* w; struct yd* n;}
```

where `hd` is `struct hd {int a, b;}`. Then, the following law is specified:

```
lf:f ==> fi AS LIST[SLL,n]
            WITH[f.@DOM ==> xd.x;
                 f.@RAN
                 ==> xd.ys AS LIST[SLL,n]
                           WITH[f.@RAN.@DOM ==> yd.y;
                                f.@RAN.@RAN.1
                                ==> yd.h AS RECORD
                                          WITH[f.@RAN.@RAN.1.a ==> hd.a;
                                               f.@RAN.@RAN.1.b ==> hd.b];
                                f.@RAN.@RAN.2 ==> yd.w
                               ]
                ]
```

Consider the following excerpt from some specification:

$[NAME]$
$AddPerson == [first?, last? : NAME \ldots \mid \ldots]$

Assume the implementation stores the first and last name of persons in a single character string variable, `name`. Then, the refinement law is as follows:

```
person:first?, last? ==> last? ++ ", " ++ first? ==> name
```

2.9 Data Structures

⟨*dataStruct*⟩ ::= `ARRAY`
| `RECORD`
| `LIST`
| `MAPPING`
| ⟨*list*⟩ | ⟨*map*⟩ | ⟨*reference*⟩ | ⟨*enumeration*⟩ | ⟨*table*⟩ | ⟨*file*⟩

The `AS` directive receives as a parameter the name of a data structure. This name represents the intended meaning of the implementation variable involved in the refinement law. Some of these parameters have their own parameters. We will explain the details in the following sections. If the intended meaning for an implementation variable is one of the included in this section, then the `AS` directive is mandatory.

2.9.1 Arrays

If an implementation variable is an array then it must be qualified by the `ARRAY` token. Arrays are considered static data structures in the sense that their length is defined at compile-time. Array indexes will start at 0 or 1 depending on the target programming language passed as parameter to the interpreter.

When a Z variable is a sequence of a simple type and it is implemented as an array, then the `WITH` directive is unnecessary:

```
l04:xs  ==> h AS ARRAY
```

because the interpreter will assign each element in the sequence to the same position in the array.

Some times we need to mention the contents of any component of an array in the implementation. This is necessary, for instance, when each component of an array is a list. The expression `x[]` is the way to do this; it can be combined with the dot notation to access members of a record: `x[].next`, for example.

2.9.2 Records

If an implementation variable is composed by more than one named variable and it is treated as a unit of storage by the programming language, then it must be qualified by the `RECORD` token. Within this category fall the `class` structure of object-oriented languages, the `struct` of the C programming language, the `record` of Pascal, etc.

The `WITH` clause can be used to specify how different components of the specification variables are refined into different members of the record. When the specification variable is of a schema type or a cross product such that: (a) all of its components are of simple types; (b) if it is a schema type, there are a member in the schema whose name coincides with the name of a field in the record; (c) if it is a cross product, it has the same number of components than fields in the record, and in this case the mapping between components and fields is positional.

If `r` is a variable of a record-type, then each of its fields can be accessed by dot notation: `r.name`.

2.9.3 Lists

⟨*list*⟩ ::= `LIST[[`⟨*listType*⟩`,(`⟨*iName*⟩ | ⟨*iName*⟩`,`⟨*iName*⟩`)]]`

⟨*listType*⟩ ::= `SLL` | `DLL` | `CLL` | `DCLL`

The list of arguments is optional. The first argument to the `LIST` token is the kind of list:

- `SLL` stands for a simply-linked list. That is, a list such that its nodes have a variable pointing to the next node. The last node of the list points to a null position. It is assumed that the node has at least one extra variable to hold the data.
- `DLL` stands for a doubly-linked list. That is, a list such that its nodes have two variables: one pointing to the next node and the other pointing to the previous node. The variable pointing to the next node of the last node of the list points to a null position, as the variable pointing to the previous node of the first node of the list. It is assumed that the node has at least one extra variable to hold the data.
- `CLL` stands for a circular-linked list. That is, a list such that its nodes have a variable pointing to the next node and the last node points to the first one. It is assumed that the node has at least one extra variable to hold the data.
- `DCLL` stands for a double-circular-linked list. That is, a list such that its nodes have two variables: one pointing to the next node and the other pointing to the previous node. The variable pointing to the next node of the last node of the list points to the first not; the variable pointing to the previous node of the first node of the list points to the last node. It is assumed that the node has at least one extra variable to hold the data.

Therefore, the lists built by Fastest for each test case will have the properties mentioned above according to the first parameter passed to the `LIST` token.

Then, the `LIST` token may have one or two more arguments. It will have one when the first argument is `SLL` or `CLL`; it will have two in any other case. If the first argument is `SLL` or `CLL`, then the second argument must be the name of the variable pointing to the next node in each node. It is assumed that there is one of these variables per node in the list. If the first argument is `DLL` or `DCLL`, then the second argument must be the name of the variable pointing to the next node in each node, and the third argument must be the name of the variable pointing to the previous node in each node. It is assumed that there is one of these variables per node in the list.

If `l` is a list whose components are of a record type, then each field of this record can be accessed by dot notation: `l.name` means the `name` field of any node in the list `l`.

2.9.4 Mappings

In general, the refinement of a specification variable as a mapping (or map) can be specified by the language described so far, as we will show shortly. However, for those languages that provide standard data structures for mappings (such as Java), the `MAPPING` directive must be used. We consider a mapping as any data structure holding a collection of *values* indexed by some *key*. Essentially, a mapping is built by adding ordered pairs of the form $(key, value)$. Conceptual examples are associative arrays and hash tables; while some concrete examples are awk's associative arrays and Java's HashMap class.

First, we will show how to specify that a specification variable must be implemented as a mapping when the implementation language does not provides a standard data structure. We will show it for the C programming language through the following examples. Say $clients : (UIDTYPE \times UID) \pfun (NAME \times ADDR)$ is a specification variable that must be implemented as a mapping. We assume that $UIDTYPE$, UID, $NAME$ and $ADDR$ are all given types that must be implemented as strings. Also, $UIDTYPE$ and UID form the key and $NAME$ and $ADDR$ is the value.

- The implementation of the mapping is an array of simply-linked lists whose nodes are records with three fields, two of which are strings and the third is a pointer to the next node. The key of an elements is obtained by applying a function named `hash` which takes two strings and returns an integer.

```
map:clients ==> hash(clients.@DOM.1 ==> k1, clients.@DOM.2 ==> k2)
                ==> c AS ARRAY
                      WITH[clients.@RAN
                           ==> c[] AS LIST[SLL,nd]
                                   WITH[clients.@RAN.1 ==> data.name;
                                        clients.@RAN.1 ==> data.addr]]
```

  In the case where `hash` takes a record of two strings (instead of two separate strings) the first part of the refinement law is:

```
map:clients ==> hash(clients.@DOM
                     ==> key AS RECORD
                             WITH[clients.@DOM.1 ==> k1,
                                  clients.@DOM.2 ==> k2])
.....................................................
```

- The implementation of the mapping is two separate arrays (each of which is of a different record type), one for the keys and the other for the values. No hash function is applied.

```
map:clients ==> ckey AS ARRAY WITH
                       [clients.@dom.1 ==> key.utype,
                        clients.@dom.2 ==> key.id];
                cval AS ARRAY WITH
                       [clients.@ran.1 ==> val.name,
                        clients.@ran.2 ==> val.addr]
```

  If a function is used to find the position to store a particular pair, then the law should be:

```
map:clients ==> hash(clients.@DOM.1 ==> k1, clients.@DOM.2 ==> k2)
                ==> ckey AS ARRAY;
                hash(clients.@DOM.1 ==> k1, clients.@DOM.2 ==> k2)
                ==> cval AS ARRAY
                         WITH [clients.@ran.1 ==> val.name,
                               clients.@ran.2 ==> val.addr]
```

Note that in this case the array for the keys should be a plain array of integers. Also, observe that `hash` must be called in both sentences.

- A simply-linked list whose nodes are records with three fields, two of which are strings and the third is a pointer to the next node. The position in the list for a particular element is obtained by applying a function named `hash` which takes two strings and returns an integer.

```
map:clients ==> hash(clients.@DOM.1 ==> k1, clients.@DOM.2 ==> k2)
                ==> c AS LIST[SLL,next]
                      WITH [clients.@ran.1 ==> val.name,
                            clients.@ran.2 ==> val.addr]
```

On the other hand, if the implementation language provides a standard data structure for mappings the refinement rule is simpler. In this case, it is assumed that this data structure stores ordered pairs whose first component is of some implementation type `K` and whose second component is of some implementation type `V`. In this case, part of the responsibility of the data structure is to determine the position where each new element is stored, so no hash function is called from the refinement law.

For example, we can consider the example developed above but this time for the Java programming language. Assume the type of the keys is a class named `UserID`, which has internal members `utype` and `uid`, and the type of the values is a class named `UserData`, which has internal members `name` and `addr`. Finally, let's say that the mapping is variable `c`. Then the refinement law is as follows:

```
map: clients ==> c AS MAPPING
                 WITH[clients.@DOM.1 ==> UserID.utype;
                      clients.@DOM.2 ==> UserID.uid;
                      clients.@RAN.1 ==> UserData.utype;
                      clients.@RAN.2 ==> UserData.uid]
```

2.9.5 References or Pointers

⟨*reference*⟩ ::= `REF[`⟨*iName*⟩`]`

Some times a variable is implemented as a pointer to other entity. In this case the implementation variable is a reference or a pointer. This character must be made explicit in the `AS` directive with the `REF` token. The argument received by this instruction is the name of the variable to which the pointer must point to. For instance, consider the specification of Figure 1. Assume *owners* is implemented as doubled-linked list, `o`, declared as:

```
struct odata {int *puid; char *pn; struct odata *n,*p;} *o;
```

where `puid` should point to the `uid` member of a node in some list `c`; `pn` should point to the `num` member of a component of some array `b`; and `n` and `p` are pointers to the next and previous nodes in the list, respectively. Then, the refinement law for *owners* would be:

```
l06:owners ==> o AS LIST[DLL,n,p] WITH[owners.@DOM ==> odata.puid AS REF[c.uid];
                                      owners.@RAN ==> odata.pn AS REF[b.num]]
```

It says, for example, that each element in the domain of *owners* is stored in the `puid` member of an `odata` node, as a pointer or reference to the `uid` member of a node in the same position but in a list `c`.

2.9.6 Enumerations

⟨*enumeration*⟩ ::=
`ENUM[[`
(⟨*sName*⟩ `==>` (⟨*iName*⟩ | ⟨*number*⟩)`{,`⟨*sName*⟩ `==>` (⟨*iName*⟩ | ⟨*number*⟩`}`)
| ⟨*number*⟩`]]`

If an specification variable is of an enumerated type (i.e. a free type without recursion), then it will usually be implemented as some enumeration. FTCRL distinguishes three alternatives:

- The variable is refined into an implementation variable of an enumerated type that has the same number of elements. In this case each of the elements at the specification level is mapped onto the element in the same position of the implementation-level enumerated type. It is assumed that at implementation level there is a language construction that clearly identifies enumerated types.

  In this case the optional argument of the `ENUM` token is not necessary.

- The variable is refined into an integer. In this case, an integer number must be passed as an argument to the `ENUM` directive (i.e. the second option). If the argument is n, then the first element of the enumerated type at specification level is mapped onto n, the second onto $n+1$ and so on.

  For example, if we have $ERRORS ::= ok \mid error1 \mid error2$ and:

  ```
  ENUM[-1]
  ```

  then, the mapping is as follows:

  ```
  ok     ==> -1,
  error1 ==>  0,
  error2 ==>  1
  ```

- Any of the conditions of the previous points does no hold. Then, the user must give the mapping between the types at both levels. This mapping is given in the optional argument of the `ENUM` token. For example:

  ```
  ENUM[ok     ==> 1,
       error1 ==> 2,
       error2 ==> 51
      ]
  ```

  At the right hand side of each `==>` token it is possible to write either an integer number or an element of an enumerated type defined in the implementation, but all of them must be of the same type.

2.9.7 Tables in Databases

⟨*table*⟩ ::= `TABLE[`⟨*iName*⟩`,`⟨*path*⟩`,`⟨*fName*⟩`]`

When a variable must be refined into a table in a database the `TABLE` modifier must be used. In this case, the interpreter will add code to each test case to access the database, create the table and fill it as specified in each test case. It is assumed that there exists a connection to the database that was established by the `init()` function defined in the preamble (see Section 2.2). The precise code that is added to interact with the database depends on the parameters with which the interpreter was invoked (see Section 4).

The first argument to the `TABLE` modifier is the name of an implementation variable through which the database is accessed. It can be a file descriptor, a socket or an object in an object-oriented programming language. It is assumed that by operating on this variable the database can be accessed. For instance, if the variable is `session` the code to create a table in the database to which `session` is connected might be something like:

```
create_table(session, "clients", ...)
```

where `create_table()` is part of an API, a framework, etc. used to access databases. It is assumed that this variable is within the scope of the concrete test case (see the specific details in the semantic section of each programming language).

The second argument is a path to the third argument which is a file name. This file is a text file containing the specification of the table to which the variable must be refined. Each column is specified in one line with the following format (tables must have at least two columns, then there must be at least to lines):

```
column_name:column_type:column_size
```

The interpreter will add code to each test case so that: (a) if the table does not exists in the database, it is created; and (b) if the table does exists, it is emptied (so it only contains the rows specified in the test case).

If the `TABLE` modifier is used then a `WITH` directive must be included except when the specification variable is of a schema type or a cross product such that: (a) all of its components are of simple types; (b) if it is a schema type, there are a member in the schema whose name coincides with the name of a column in the table; (c) if it is a cross product, it has the same number of components than columns in the table, and in this case the mapping between components and columns is positional—so watch the specification of the table in the text file.

If none of the preceding conditions holds, a `WITH` clause must be included so the relation between parts of the specification variable and columns in the table is specified. For example, assume there is a state variable $clients : UID \pfun NAME \times ADDRESS \times \mathbb{N}$ where UID, $NAME$ and $ADDRESS$ are basic types. Say this variable must be refined into a table called `clientData` specified as follows:

```
cid:int:10
age:int:2
name:char:40
addr:char:40
```

Therefore, the refinement law would be:

```
tbl1:clients ==> clientData AS TABLE[connection, /home/usr, clientData.txt]
                            WITH[clients.@DOM   ==> clientData.cid;
                                 clients.@RAN.3 ==> clientData.age;
                                 clients.@RAN.1 ==> clientData.name;
                                 clients.@RAN.2 ==> clientData.addr]
```

2.9.8 Files in File Systems

⟨*file*⟩ ::= `FILE[`⟨*path*⟩`]`

When a variable must be refined into a file in a file system the `FILE` modifier must be used. In this case, the interpreter will add code to each test case to create the file and fill it as specified in each test case. The precise code that is added to interact with the file depends on the parameters with which the interpreter was invoked (see Section 4).

The argument to the `FILE` modifier is a path to the file. Currently, only text files are supported. The interpreter will add code to each test case so that: (a) if the file does not exists in the path, it is created; and (b) if the file does exists, it is emptied (so it only contains what is specified in the test case).

If the `FILE` modifier is used and it does not include a `WITH` directive, then the expression at the left of the closest `==>` token, is transformed into a string and it is copied to the file. The transformation follows the same rules of the string expressions described in Section 2.8.

As a first example consider the following refinement. Assume there is a state variable *clients* : $UID \pfun NAME \times ADDRESS \times \mathbb{N}$ where *UID*, *NAME* and *ADDRESS* are basic types. Say this variable must be refined into a file named `clientData` with one record per row with the following format:

```
UID:ADDRESS:AGE-NAME
```

Therefore, the refinement law would be:

```
file:clients ==>
      clients.@DOM ++ ":"
      ++ clients.@RAN.2 ++ ":"
      ++ clients.@RAN.3 ++ "-"
      ++ clients.@RAN.1 ==> clientData.txt AS FILE[/home/usr]
```

2.10 User Input

If a specification variable corresponds to input provided by the user, the adaptation sentence is:

```
specVar ==> @UINPUT
```

where `@UINPUT` is introduced in the language because user input is read by means of a system call before being stored in an implementation variable.

If a subroutine reads input from the user several times during a run, then the specification should declare a set variable whose elements represent each and every input of the subroutine—if order matters then the set should be a sequence, which is also a set [**?**, Chapter 4]. As always the set is processed element by element and the result of translating each of them will be provided as input as the SUT needs them.

If the subroutine waits a fixed number of inputs from the user—like in a form—but this has been abstracted away in the specification in a single, ground-typed variable, then the following sentence solves the problem:

```
specVar ==> @AUTOFILL ==> @UINPUT;
            @AUTOFILL ==> @UINPUT;
            @AUTOFILL ==> @UINPUT;
```

If the inputs expected by the program have been abstracted in different specification variables and the order in which they are read matters, then the following sentence is the right one:

```
v1, v2, v3 ==> v2 ++ "\n" ++ v3 ++ "\n" ++ v1 ==> @UINPUT;
```

2.11 System Date

If a specification variable corresponds to the system date, the adaptation sentence is:

```
specVar ==> @SYSDATE
```

where `@SYSDATE` is introduced in the language because the system date is read by means of a system call before being stored in an implementation variable.

If a subroutine needs the system date several times during a run, then the specification should declare a set variable whose elements represent each and every date that the subrutine should read—if order matters then the set should be a sequence, which is also a set [**?**, Chapter 4]. As always the set is processed element by element and the result of translating each of them will be provided as input as the SUT needs them.

2.12 Identifiers, Names and Other Minor Syntactical Elements

⟨*name*⟩ ::= ⟨*letter*⟩{_ | ⟨*digit*⟩ | {⟨*name*⟩}}

⟨*sName*⟩ ::= valid Z identifier

⟨*iName*⟩ ::= ⟨*iIdent*⟩ | ⟨*iIdent*⟩`[]` | ⟨*iIdent*⟩`.`⟨*iIdent*⟩ | ⟨*regExpr*⟩

The ⟨*iIdent*⟩`[]` expression in the ⟨*iName*⟩ production is valid only when ⟨*iIdent*⟩ is an array or a list. ⟨*iIdent*⟩`.`⟨*iIdent*⟩ and ⟨*iIdent*⟩`.*` are valid when the left ⟨*iIdent*⟩ is a type name of a record or a table structure. The last three alternatives of ⟨*iIdent*⟩ can be used only inside a `WITH` directive.

⟨*iIdent*⟩ ::= valid identifier in the programming language

⟨*fName*⟩ ::= valid identifier of a file in the operating system

⟨*path*⟩ ::= valid path in the Linux operating system

⟨*string*⟩ ::= any charcter string enclosed in double quotes

⟨*setExtension*⟩ ::= any valid Z set extension

⟨*PLCode*⟩ ::= legal text of some programming laguage

⟨*digit*⟩ ::= `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

⟨*number*⟩ ::= ⟨*digit*⟩{`0` | ⟨*digit*⟩}

⟨*letter*⟩ ::= `a` | . . . | `z` | `A` | . . . | `Z`

3 Common Semantics

In this section we describe some semantics rules of FTCRL with respect to an abstract programming language (APL). In later sections, we will show how to map the rules given here for each real programming language. However, if some of the features of our abstract programming language cannot be mapped onto some real programming language, then this programming language cannot be supported by Fastest.

Our first step is, then, to define the syntax and semantics of this APL. Fortunately, we only need to define how variables are declared, initialized and assigned. Secondly, we will show what sentences in this APL are going to be generated for a refinement rule.

3.1 The Abstract Programming Language (APL)

Our abstract programming language has the following features.

- A variable is declared by giving its name and type.

  (If this is not mandatory in a real programming language, then FTCRL will work only for those programs where variables have been declared as we assume.)

- Types include:
  - Integer numbers.
  - Floating point numbers. Any integer number is also a floating point number.
  - Characters.
  - Enumerations.
  - Arrays. We consider them to be static, i.e. their sizes are defined at compile time.
  - Records. That is, a data structure that stores simultaneously two or more values each of which is bound to a named internal variable, called member.
  - Lists. That is, a data structure whose size can be modified at runtime.
  - References. That is, a variable that can point or refer to another variable.
- These types can be recursively defined to form complex data structures.
- We assume that when a variable is declared it can be initialized.
- Variable declaration is written as follows (the initialization part is optional):
  - For variables whose type is `int`, `float` or `char`:

    ```
    type var = expr;
    ```

    where `expr` must be a constant expression.
  - For enumerations:

    ```
    name enum [const1, ..., constN] a = constI;
    ```

    where `name` is the name for the enumeration-type being declared; this name can be used later to declare more variables, arrays or lists; `const1, ..., constN` are `N` distinct identifiers that are not used anywhere else in the program; and `constI` is one of these identifiers.
  - For arrays:

```
type a[size] = [expr1, ..., exprSIZE];
```

where `size` is the size of the array and `expr1, ..., exprSIZE` are `size` constant expressions.

– For records:

```
name record[type1 mem1, ..., typeN memN] var =
                                    [mem1 = expr1, ..., memN = exprN];
```

where `name` is the name for the record-type being declared; this name can be used later to declare more variables, other record-types, arrays or lists; and `expr1, ..., exprN` are `N` constant expressions.

Alternatively, records can be initialized by using the dot notation:

```
name record[type1 var1, ..., typeN varN] var;
var.mem1 = expr1;
... ;
var.memN = exprN;
```

– For lists (in general they can be initialized only with the empty list):

```
name list[type] var = [];
```

where `name` is the name for the list-type being declared; this name can be used later to declare more variables, records, arrays or other list-types; and `[]` is the empty list for any type.

The only list-type that can be initialized to a non-empty list is `list[char]`:

```
string list[char] s = "abcde234-()"
```

We will denote `list[char]` by `string` since programming languages usually treat them differently from lists of other types.

Prior to be able to add an element to a list it is necessary to ask for space:

```
s = mem(type, n);
```

where `s` is a list of type `list[type]`, `mem()` is a system call that allocates memory space and `n` is the number of components to be allocated. Once the list has space, it can be assigned as follows: `s[i] = expr` where `expr` is an expression of type `type`.

– For references:

```
type ref s = &var
```

where `type` is the type of `var` which is a variable already declared—including members of record-types. The symbol `&` makes `s` to refer or point to `var`. If `var` has the same type of `s`, then `s = var` means that `s` refers or points to the same variable than `var`. There is no other way to assign something to a reference.

Programming languages usually feature many other operators to work with references but within our context this is enough.

– For instance these are all valid recursive declarations:

```
lINT list[int] xs;

colors enum [red, blue, green] c = green;

myRecord record[int a, lINT z, colors h] w = [a = 2, z = [], h = green];

myRecord y[3] = [[a = 2, z = [], h = bule], [a = 5, z = [], h = blue]];
```

– It is also valid to declare types as follows:

```
name enum [const1, ..., constN];

name record[type1 var1, ..., typeN varN];

name list[type];
```

and then use each `name` to declare variables.

- The assignment sentence is equal to the initialization part of each declaration shown above.

3.2 Semantic Rules

We show the semantic rules for each Z type, from the simplest to the most complex. At the end, we show how this semantics rules must be combined to deal with FTCRL recursion.

3.2.1 Preprocessing

Each refinement rule is preprocessed before the code generation stage begins. Only refinement rules including `@LAWS` and `@UUT` sections are considered, but in this case all the refinement rules referenced from the former are also considered. In other words, the interpreter only pays attention to refinement rules including `@LAWS` and `@UUT` sections. During preprocessing a new refinement rule is (internally) written replacing by making the following modifications:

1. References to other refinement rules are replaced by their texts.
2. Variable substitutions are performed where indicated (cf. $\langle varSubst \rangle$).

According to the grammar presented in the previous section, a refinement law can be of the following form:

v, w, x, y, z `==>`
  (v, w `==>`
    v `==>` $\langle refinement \rangle_v$;
    w `==>` $\langle refinement \rangle_w$;
)
  (x, y, z `==>`
    x `==>` $\langle refinement \rangle_x$;
    y, z `==>` $\langle refinement \rangle_{y,z}$;
)

Only sentences ending in a $\langle refinement \rangle$ non-terminal make FTCRL to produce code in the implementation programming language. All the other sentences can be elimated producing the following refinement law:

v `==>` $\langle refinement \rangle_v$;
w `==>` $\langle refinement \rangle_w$;
x `==>` $\langle refinement \rangle_x$;
y, z `==>` $\langle refinement \rangle_{y,z}$

Furthermore, sentences of the form v `==>` $\langle refinement \rangle_v$ where $\langle refinement \rangle_v$ is of the form:

v `==>` $\langle iExprRefinement \rangle$

thus making the original sentence be of the form:

v `==>` v `==>` $\langle iExprRefinement \rangle$

because the second v corresponds to the $\langle sExprRefinement \rangle$ of the $\langle refinement \rangle$ expression, are changed to:

v `==>` $\langle iExprRefinement \rangle$

Besides, in sentences of the form y, z `==>` $\langle refinement \rangle_{y,z}$, the $\langle refinement \rangle$ expression must be of the form:

$\langle sExprRefinement \rangle_{y,z}$ `==>` $\langle iExprRefinement \rangle$

where $\langle sExprRefinement \rangle_{y,z}$ must be an expression depending on both y and z (i.e. it cannot be a single variable because in this case there would be a variable listed at the left hand side but not used at the right hand side; this is allowed only when the left hand side lists only one variable). In this last case, FTCRL replaces the sentence (y, z `==>` $\langle refinement \rangle_{y,z}$) as follows:

$\langle properSExprRefinement \rangle_{y,z}$ `==>` $\langle iExprRefinement \rangle$

where $\langle properSExprRefinement \rangle$ is a $\langle sExprRefinement \rangle$ which is not a variable.

Then, at this point refinement sentences are of one of the following forms:

$\langle sName \rangle$ `==>` $\langle iExprRefinement \rangle$
$\langle properSExprRefinement \rangle$ `==>` $\langle iExprRefinement \rangle$

Now FTCRL binds a type to the left hand side of each sentence as follows:

- If the left hand side is a (specification) variable, then it is the Z type of the variable.
- If the left hand side is a proper $\langle sExprRefinement \rangle$ (i.e. it is not a single variable), then the type is the type of this expression as follows:
  - $\langle zExprSet \rangle$. In this case the expression always have a Z type because it is either a Z set expression or a Z variable. So the type is this Z type.
  - $\langle zExprNum \rangle$. In this case the type is $\mathbb{Z}$.
  - $\langle zExprString \rangle$. If the expression produces just one value, then the type is called "FTCRL's String Type" or FST; if it produces a set of values then the type is $\mathbb{P}$ FST.
  - $\langle zExprSeq \rangle$. The type is the Z type of the expression.
  - $\langle funAppExpr \rangle$. If one or more of the arguments of the function call is of a set type, then the type of the left hand side is $\mathbb{P}$ `X` where `X` is the implementation type returned by the function call. Otherwise the the type of the left hand side is the implementation type returned by the function call.

1: **for** $t \in ATC$ **do**
2: $\quad rr \leftarrow rr[\forall\, v : \mathrm{dom}\, t \bullet v/t(v)]$
3: $\quad rr \leftarrow rr[\forall\, e : expr(rr) \bullet e/eval(e)]$
4: $\quad generateCode(rr)$
5: **end for**

Figure 4: Interpretation algorithm. ATC is the list of abstract test cases received as a parameter; rr is the refinement rule.

In summary, at this point, the left hand side of a refinement law has one of the following types:

- A Z type
- The FTCRL's String Type
- An implementation type
- A type of the form $\mathbb{P}\,\mathtt{X}$ where `X` is an implementation type

It is worth to be mentioned that refinement sentences inside a `WITH` directive or as arguments of a function call, are $\langle refinement \rangle$ non-terminals. Therefore, they are treated as explained above.

3.2.2 Processing the List of Abstract Test Cases

The interpreter receives the definition of a refinement rule and a list of abstract test cases written in LaTeX markup. It applies the refinement rule to each abstract test case, thus generating a list of concrete test cases. Each concrete test case is a program in our APL (i.e. it can be thought as a text file containing a sequence of instructions of the APL). We call *current test case* to the abstract test case that is being processed at each moment.

The interpreter runs the abstract algorithm shown in Figure 4. ATC is the list of abstract test cases received as a parameter; rr is the refinement rule. Here we think of each $t \in ATC$ as a function from its set of variables into the corresponding values: $t : VAR \to VAL$. Then, $t(v)$ is the value of v according to t's definition. The notation $rr[\forall\, v : \mathrm{dom}\, t \bullet v/t(v)]$ means substituting each variable of t appearing in rr by its value. Then, after step 2 all the expressions at the left hand side of a refinement sentence, are constant expressions. $expr(rr)$ is the set of expressions (i.e. non terminals) at the left hand side of a refinement sentence in rr. Then, in step 3 all the elements of $expr(rr)$ are substituted by their values. Step 3 is sound since a concrete test case is no more than initializing all the UUT variables and calling it. In step 4 the interpreter starts to generate the code that forms the concrete test case corresponding to the current test case.

3.2.3 The `@PREAMBLE` Section and Beginning the Concrete Test Case

The contents of the preamble is the first part of each concrete test case. Therefore, in a way or another at the beginning of each concrete test case there is the declaration of every implementation variable and function mentioned in the `@LAWS` section except, perhaps, for the names of the parameters appearing in the `@UUT` section, if they were not declared in the preamble. This includes the definition of the types of each variable, if necessary. The interpreter might declare local, unused variables as it process abstract test cases and laws.

Since a priori we do not know whether it is necessary to declare a type and a variable, the semantics rules we are going to explain below include source code in the APL for declaring variables. The interpreter will check during the code generation phase whether it is necessary to include the declaration or not. If not, then the declaration part of the semantic rule will be omitted.

The preamble must contain the definition of a function called `init()` (see Section 2.2). If this name conflicts with another name (introduced in the preamble) then the interpreter must change it for a new, unused name. The following semantic rule must, then, take this into consideration. Right after the contents of the preamble the interpreter writes the following while generating the concrete test case:

```
begin main()                                                    (1)
if !init() then print("Initialization error"); return 0; endif
```

where `!` means negation and it is assumed that when the resulting concrete test case is compiled, execution begins at `main()`.

3.2.4 The Declarations, Assignments and Closing Lists

The interpreter builds each concrete test cases by concatenating three lists of APL instructions. The first list, known as the *declarations list* (*DL*) is a list of variable declarations added by the interpreter. The second list, known as the *assignments list* (*AL*) is a list of assignments as defined in the APL. The third list, known as the *closing list* (*CL*) is a list of APL instructions for management of external resources such as files and tables. *DL* is concatenated right after (1), *AL* right after *DL* and *CL* right after *DL*, when the last refinement law is processed.

Therefore, the declarations and the assignments listed in the following semantics rules must be added to *DL* and *AL*, respectively. However, recall that some declarations mentioned in the semantic rules may be omitted since they are already available in the preamble. Furthermore, in many semantic rules we give a declaration along with an initialization: `type var = val;`. In these cases, the declaration part, `type var;` is added to *DL*, while the initialization part, `var = val;` is added to *AL*. Instructions for *CL* are explicitly mentioned in the semantics rules.

3.2.5 Processing Order

Given a refinement rule, its refinement laws are processed in any order. Given a refinement law, its sentences are processed in any order. Given a refinement sentence, its ⟨*refinement*⟩ non-terminals are processed from the innermost `WITH` directive to the outermost; next, the ⟨*refinement*⟩ non-terminals belonging to the argument list of a function call are processed; finally, sentences in the APL liking all the APL sentences generated in the previous steps are generated.

The only exception to that processing order is when a refinement law r_1 contains a `REF` directive to an implementation variable in refinement law r_2. In this case, r_2 is processed before r_1.

3.2.6 Record types referenced inside a `WITH` directive

As we have said in earlier sections, it is possible to use dot notation inside a `WITH` directive to access components of a record. In effect, is `T` is the name of a record type and `n` is one of its components, then the refinement specification inside a `WITH` directive may contain something of the form `... ==> T.n`. Every time such a refinement specification is processed a new variable of type `T` is created and used as the target for the refinement. Since sets are processed one element at a time, then such a refinement specification will generate as many new variables as elements are in the set.

Note that if the refinement specification within a `WITH` directive contains more than one refinement sentence whose right hand side is of the form `T.n` for some field `n`, then the same variable of type `T` is used for all the sentences. For example, for the following refinement law:

```
... WITH[... ==> T.n; ... ==> T.m]
```

a new variable of type `T` will be created (every time the `WITH` directive is processed) and it will be used to set the values for both `n` and `m`.

3.2.7 Refining the result of a function call

When the left hand side of a refinement sentence is a call to an implementation function, FTCRL proceeds as follows. If there are more than one call to the same function with the same arguments in a refinement law, then only one call is made every time the law is processed, the result is stored in a variable and that variable is used in those refinement sentences where the call is present. This is to avoid possible side effects when two ore more calls are made. So for now on, we can think that there is one function call in a refinement law.

The refinement sentences passed as arguments to the function call are all processed before placing the call. These refinement sentences obey the rules described in the following sections. In doing so, the result of refining each argument yields an initialized variable of the type expected by the function as an argument in that position. Say these variables are `a_1, ..., a_m`. Also a variable of the type returned by the function is declared. Say this variable is `r`. Then a sentence of the following form is generated (after the sentences that initialize `a_1, ..., a_m`):

```
r = f(a_1, ..., a_m);
```

If one ore more of the left hand sides of the arguments of the function call is of a set type, then the procedure described so far is executed for each element of the set and some other considerations apply. In this case, if this set has n elements in a given test case, then n sets of `a_1, ..., a_m, r` must be defined and used in each call. All this is explained in Section 3.2.16.

Otherwise, the semantics is very simple because the value returned by the function is used to initialize the implementation variable at the right hand side of the refinement sentence. Clearly, in this case the type returned by the function must be compatible, according to the type rules of the target programming language, with the type of the implementation variable at the right hand side.

3.2.8 Regular expressions as the target of a refinement rule

As shown when the grammar was introduced, the ⟨*iName*⟩ production includes regular expressions (⟨*regExpr*⟩). Therefore, it is possible to have something of the form `... ==> regExpr`. If this is the case then all implementation variables whose name matches the regular expression will be initialized by this refinement rule. If the left hand side of the refinement rule represents n values (for example a set) and there are m variables matching the regular expression then we have:

- $n < m$: each of the m variables receives one of the values; all the values must be used and some will be used more than once.
- $n = m$: each variable receives one value; all values must be used.
- $n > m$: each variable receives one value; m values must be used.

3.2.9 User Input

The semantics of the `@UINPUT` directive is as follows:

- If the type of *specVar* is a ground type, then its value is translated as a character string, by following the rules described below;
- If the type of *specVar* is a set, then each of its values is translated as a character string, as above;
- The result of translating *specVar* is stored in a text file one character string per line;
- Each test script is executed by redirecting the input from the file created in the previous step—how the tool do this depends on the operating system passed as parameter to the interpreter, see Section 4.

3.2.10 System Date

The semantics of the `@SYSDATE` directive depends on the concrete implementation language. Essentially, the interpreter must intercept calls to the function returning the system date and return, instead, the date values indicated in the test case. The interception can be done by declaring a function with the same signature of the function used by the language to get the system date but whose implementation returns the list of dates specified in the test case.

We will give the semantics rules of FTCRL starting from step 4 of Figure 4 by describing the APL sentences that are generated when constants whose types are one of the types listed in Section 3.2.1 must be refined into some implementation variable. We concentrate on type (and not in variables or expressions) since after step 3 only typed constants are left to be refined.

In the semantics rules, `X` (i.e. in upper case typewriter type) denotes the value of the expression at the left hand side of the `==>` token. Note that X (i.e. in italic type) might denote the Z type of `X`.

3.2.11 $\mathbb{Z}$

From the FTCRL perspective, it only make sense to refine a $\mathbb{Z}$ left hand side as described by any of the following refinement expressions:

Simple Types

- $\mathbb{Z}$ `==> a` where `a` is a variable of type `int`, then the following sentence in the APL will be generated for the current test case[3]:

```
int a = X;
```
(2)

- $\mathbb{Z}$ `==> a` where `a` is a variable of type `float`, then the following sentence in the APL will be generated for the current test case:

```
float a = X;
```
(3)

- $\mathbb{Z}$ `==> a` where `a` is a variable of type `char`, then `X` is converted into the character whose ASCII code is `X`. We denote this value by `"cX"`. Then the following sentence in the APL will be generated for the current test case:

```
char a = "cX";
```
(4)

- $\mathbb{Z}$ `==> a` where `a` is a variable of type `name enum [const1, ..., constN]`, then `X` is converted into the X^{th} element of `name`. We denote this value by `constX`. Then, the following sentence in the APL will be generated for the current test case[4]:

```
name enum [const1, ..., constN];
name a = constX;
```
(5)

- $\mathbb{Z}$ `==> a` where `a` is a variable of type `string`, then the following sentence in the APL will be generated for the current test case:

```
string a = "X";
```
(6)

[3]With respect to what was said in Section 3.2.3, if `a` is already declared, then the interpreter will generate just `a = X;`. We will not mention this any further.

[4]With respect to what was said in Section 3.2.3, if `name` is already declared, then the interpreter will generate just `name a = constX;`, and if `a` is also already declared, then it will generate just `a = constX;`. We will not mention this any further.

References

- $\mathbb{Z}$ `==> a AS REF[b]` where `a` is a variable of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, and `b` is of type `type`. Then the following sentence in the APL will be generated for the current test case:

  ```
  type a = &b;
  ```
  (7)

  Recall that each concrete test case is built by concatenating DL with AL and that the previous sentence is divided into `type a;`, which is added to DL and `a = &b;`, which is added to AL. Then `a = &b` is always after `b`'s declaration no matter whether the refinement specification including `b` is processed before or after the current one. This reasoning is valid for all uses of the `REF` clause, we will not repeat it any further.

- $\mathbb{Z}$ `==> a AS REF[b]` where `a` and `b` are variables of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`. Then the following sentence in the APL will be generated for the current test case:

  ```
  type a = b;
  ```
  (8)

- $\mathbb{Z}$ `==> a AS REF[r.b]` where `a` is a variable of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, `r` is some record such that `b` of type `type` is one of its members. Then the following sentence in the APL will be generated for the current test case:

  ```
  type a = &r.b;
  ```
  (9)

- $\mathbb{Z}$ `==> a AS REF[r.b]` where `a` is a variable of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, `r` is some record such that `b` of type `type ref` is one of its members. Then the following sentence in the APL will be generated for the current test case:

  ```
  type a = r.b;
  ```
  (10)

Members of a Record-Type

- $\mathbb{Z}$ `==> r.a AS REF[b]` where `r` is some record such that `a` of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, and `b` is of type `type`. Then the following sentence in the APL will be generated for the current test case:

  ```
  r.a = &b;
  ```
  (11)

- $\mathbb{Z}$ `==> r.a AS REF[b]` where `r` is some record such that `a` of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, and `b` is of type `type ref`. Then the following sentence in the APL will be generated for the current test case:

  ```
  r.a = b;
  ```
  (12)

- $\mathbb{Z}$ `==> r.a` where `r` is some record such that `a` of type `type`, where `type` is any of `int`, `float`, `char`, `enum` or `string`, is one of its members. Then the following sentence in the APL will be generated for the current test case:

  ```
  r.a = XX;
  ```
  (13)

  where `XX` is one of `X`, `"cX"`, `constX` or `"X"` depending on the type of `a` and applying one of (2)-(6), accordingly.

Columns of Tables and Files

- $\mathbb{Z}$ `==> t.a` where `t` is some table such that `a` of type `type`, where `type` is any of `int`, `float`, `char`, `enum` or `string`, is one of its columns. Then the following sentence in the APL will be added to AL for the current test case:

  ```
  insert(c, t, ..., "a = XX", ...)
  ```
  (14)

  where `XX` is one of `X`, `"cX"`, `constX` or `"X"` depending on the type of `a` and applying one of (2)-(6), accordingly. `insert()` is a system call that inserts a record in a table stored in a database accessible through some connection `c`. For this sentence to work it is necessary to provide an argument of type `string` of the form `"col = val"`, where `col` is the name of a column of `t` and `val` is its value, for each and every column in `t`. Therefore, the interpreter will build this APL sentence as processes the entire `WITH` directive of a `TABLE` clause, which is expected to be complete with respect to the table definition.

  Besides, the following sentence will be added to DL:

  ```
  delete(c, t);
  ```

  where `delete()` is a system call that deletes all the records stored in table `t` which is stored in a database accessible through some connection `c`. In this way the table is deleted only once at the beginning of the concrete test case.

- $\mathbb{Z}$ `==> a AS FILE[...]` where `a` is a file name. Then the following sentence in the APL will be added to DL for the current test case:

  ```
  f = open(a);
  ```
  (15)

  in this way the file is opened and emptied at the beginning of the test case. Besides, the following sentence in the APL will be added to AL for the current test case:

  ```
  write(f, "X");
  ```
  (16)

  Finally, the following sentence in the APL will be added to CL for the current test cases:

  ```
  close(f);
  ```
  (17)

3.2.12 Basic or Given Types

From the FTCRL perspective, it only make sense to refine an expression whose type is a basic type X as described by any of the following refinement expressions:

Simple Types

- X `==> a` where `a` is a variable of type `int`, then `X` is converted into an integer by applying a bijection between the set of values of type X participating in the list of abstract test cases being processed and a subset of $\mathbb{Z}$. This bijection is called it $bij_{X,\mathbb{Z}}$ and the result of applying it to `X` is denoted by `iX`

  After applying $bij_{X,\mathbb{Z}}$ to `X` the following sentence in the APL will be generated for the current test case:

  ```
  int a = iX;
  ```
  (18)

- X `==> a` where `a` is a variable of type `float`, then `X` is converted into an integer by applying $bij_{X,\mathbb{Z}}$ to it. Then the following sentence in the APL will be generated for the current test case:

```
float a = iX;
```
(19)

- X `==> a` where `a` is a variable of type `char`, then `X` is converted into an integer by applying $bij_{X,\mathbb{Z}}$ to it, and then `iX` is converted into a character as in (4). Then the following sentence in the APL will be generated for the current test case:

```
char a = "ciX";
```
(20)

- X `==> a` where `a` is a variable of type `name enum [const1, ..., constN]`, then `X` is converted into an integer by applying $bij_{X,\mathbb{Z}}$ to it, and then `iX` is converted into an element of `name` as in (5). Then the following sentence in the APL will be generated for the current test case:

```
name enum [const1, ..., constN];
name a = constiX;
```
(21)

- X `==> a` where `a` is a variable of type `string`, then the following sentence in the APL will be generated for the current test case:

```
string a = "X";
```
(22)

- X `==> a` where `a` is a variable of type `name record[...]`, then the following sentence in the APL will be generated for the current test case:

```
name record[...]  a = [rX]
```
(23)

  where `rX` is the result of applying the bijection $bij_{X,name}$ to `X`. $bij_{X,name}$ is a bijection between the set of values of type X participating in the list of abstract test cases being processed and a subset of `name`.

References

- X `==> a AS REF[b]` where `a` is a variable of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, and `b` is of type `type`. Then the following sentence in the APL will be generated for the current test case:

```
type a = &b;
```
(24)

- X `==> a AS REF[b]` where `a` and `b` are variables of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`. Then the following sentence in the APL will be generated for the current test case:

```
type a = b;
```
(25)

- X `==> a AS REF[r.b]` where `a` is a variable of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, `r` is some record such that `b` of type `type` is one of its members. Then the following sentence in the APL will be generated for the current test case:

```
type a = &r.b;
```
(26)

- X `==> a AS REF[r.b]` where `a` is a variable of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, `r` is some record such that `b` of type `type ref` is one of its members. Then the following sentence in the APL will be generated for the current test case:

```
type a = r.b;
```
(27)

Members of a Record-Type

- *X* `==> r.a AS REF[b]` where `r` is some record such that `a` of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, and `b` is of type `type`. Then the following sentence in the APL will be generated for the current test case:

```
r.a = &b;
```
(28)

- *X* `==> r.a AS REF[b]` where `r` is some record such that `a` of type `type ref` where `type` is any of `int`, `float`, `char`, `enum` or `string`, and `b` is of type `type ref`. Then the following sentence in the APL will be generated for the current test case:

```
r.a = b;
```
(29)

- *X* `==> r.a` where `r` is some record such that `a` of type `type`, where `type` is any of `int`, `float`, `char`, `enum` or `string`, is one of its members. Then the following sentence in the APL will be generated for the current test case:

```
r.a = XX;
```
(30)

  where `XX` is one of `iX`, `"ciX"`, `constiX` or `"X"` depending on the type of `a` and applying one of (18)-(22), accordingly.

Columns of Tables and Files

- *X* `==> t.a` where `t` is some table such that `a` of type `type`, where `type` is any of `int`, `float`, `char`, `enum` or `string`, is one of its columns. Then the following sentence in the APL will be added to *AL* for the current test case:

```
insert(c, t, ..., "a = XX", ...)
```
(31)

  where `XX` is one of `iX`, `"ciX"`, `constiX` or `"X"` depending on the type of `a` and applying one of (18)-(22), accordingly.

  Besides, the following sentence will be added to *DL*:

```
delete(c, t);
```

  Same considerations of (14) apply for the rest of the elements of these sentences.

- *X* `==> a AS FILE[...]` where `a` is a file name. Then the following sentence in the APL will be added to *DL* for the current test case:

```
f = open(a);
```
(32)

  in this way the file is opened and emptied at the beginning of the test case. Besides, the following sentence in the APL will be added to *AL* for the current test case:

```
write(f, "X");
close(f);
```
(33)

  Finally, the following sentence in the APL will be added to *CL* for the current test case:

```
close(f);
```
(34)

3.2.13 Enumerated Types

Enumerated types are free types defined without using recursion—recursive free types are not currently supported by Fastest. From the FTCRL perspective, it only make sense to refine an expression whose type is a enumerated type $E ::= Const_1 \mid \ldots \mid Const_n$, as described by any of the following refinement expressions. In the following rules we will assume that `X` is equal to $Const_k$ for some $k \in 1 \,..\, n$.

Simple Types

- E `==> a AS ENUM` where `a` is a variable of type[5] `name enum [const1, ..., constN]`, then the following sentence in the APL will be generated for the current test case:

```
name enum [const1, ..., constN];
name a = constK;
```
(35)

- E `==> a AS ENUM[` $Const_1$ `>` $const_{j_1}$ `, ... ,` $Const_n$ `>` $const_{j_n}$ `]` where `a` is a variable of type `name enum [const1, ..., constN]` and $const_{j_1}, \ldots, const_{j_n}$ is any permutation of `const1, ..., constN`, then the following sentence in the APL will be generated for the current test case:

```
name enum [const1, ..., constN];
name a = constJK;
```
(36)

  where `constJK` is $const_{j_k}$.

- E `==> a AS ENUM[m]` where `a` is a variable of type `int`, then the following sentence in the APL will be generated for the current test case:

```
int a = k + m - 1;
```
(37)

  Note that in the actual code generated by the interpreter, `k` and `m` are numbers, not variable names.

- E `==> a AS ENUM[` $Const_1$ `>` j_1 `, ... ,` $Const_n$ `>` j_n `]` where `a` is a variable of type `int` and $j_1, \ldots, j_n$ are n different integer numbers, then the following sentence in the APL will be generated for the current test case:

```
int a = Jk;
```
(38)

  where `Jk` is j_k.

- E `==> a AS ENUM[m]` where `a` is a variable of type `char`, then $k + m - 1$ is converted into a character as in 4, say this character is denoted by `"ckX"`. Then the following sentence in the APL will be generated for the current test case:

```
char a = "ckX";
```
(39)

- E `==> a AS ENUM[` $Const_1$ `>` j_1 `, ... ,` $Const_n$ `>` j_n `]` where `a` is a variable of type `char` and $j_1, \ldots, j_n$ are n different integer numbers, then j_k is converted into a character as in (4), say this character is denoted by `"cjkX"`. Then the following sentence in the APL will be generated for the current test case:

```
char a = "cjkX";
```
(40)

[5] In these rules we take `N` $= n$ and `K` $= k$.

- *E* `==> a` where `a` is a variable of type `string`, then the following sentence in the APL will be generated for the current test case:

  ```
  string a = "ConstK";
  ```
  (41)

  Remember that in these rules `X` is equal to $Const_k$.

References

- *E* `==> a AS REF[b]` where `a` is a variable of type `type ref` where `type` is any of `enum`, `int`, `char` or `string`, and `b` is of type `type`. Then the following sentence in the APL will be generated for the current test case:

  ```
  type a = &b;
  ```
  (42)

- *E* `==> a AS REF[b]` where `a` and `b` are variables of type `type ref` where `type` is any of `enum`, `int`, `char` or `string`. Then the following sentence in the APL will be generated for the current test case:

  ```
  type a = b;
  ```
  (43)

- *E* `==> a AS REF[r.b]` where `a` is a variable of type `type ref` where `type` is any of `enum`, `int`, `char` or `string`, `r` is some record such that `b` of type `type` is one of its members. Then the following sentence in the APL will be generated for the current test case:

  ```
  type a = &r.b;
  ```
  (44)

- *E* `==> a AS REF[r.b]` where `a` is a variable of type `type ref` where `type` is any of `enum`, `int`, `char` or `string`, `r` is some record such that `b` of type `type ref` is one of its members. Then the following sentence in the APL will be generated for the current test case:

  ```
  type a = r.b;
  ```
  (45)

Members of a Record-Type

- *E* `==> r.a AS REF[b]` where `r` is some record such that `a` of type `type ref` where `type` is any of `enum`, `int`, `char` or `string`, and `b` is of type `type`. Then the following sentence in the APL will be generated for the current test case:

  ```
  r.a = &b;
  ```
  (46)

- *E* `==> r.a AS REF[b]` where `r` is some record such that `a` of type `type ref` where `type` is any of `enum`, `int`, `char` or `string`, and `b` is of type `type ref`. Then the following sentence in the APL will be generated for the current test case:

  ```
  r.a = b;
  ```
  (47)

- *E* `==> r.a` where `r` is some record such that `a` of type `type`, where `type` is any of `enum`, `int`, `char` or `string`, is one of its members. Then the following sentence in the APL will be generated for the current test case:

  ```
  r.a = XX;
  ```
  (48)

  where `XX` is one of `constK`, `constJK`, `k + m - 1`, `Jk`, `"ckX"`, `"cjkX"` or `"ConstK"` depending on the type of `a` and applying one of (35)-(41), accordingly.

Columns of Tables and Files

- *E* `==> t.a` where `t` is some table such that `a` of type `type`, where `type` is any of `enum`, `int`, `char` or `string`, is one of its columns. Then the following sentence in the APL will be added to *AL* for the current test case:

  ```
  insert(c, t, ..., "a = XX", ...)
  ```
  (49)

  where `XX` is one of `constK`, `constJK`, `k + m - 1`, `Jk`, `"ckX"`, `"cjkX"` or `"ConstK"` depending on the type of `a` and applying one of (35)-(41), accordingly.

  Besides, the following sentence will be added to *DL*:

  ```
  delete(c, t);
  ```

  Same considerations of (14) apply for the rest of the elements of these sentences.

- *E* `==> a AS FILE[...]` where `a` is a file name. Then the following sentence in the APL will be added to *DL* for the current test case:

  ```
  f = open(a);
  ```
  (50)

  in this way the file is opened and emptied at the beginning of the test case. Besides, the following sentence in the APL will be added to *AL* for the current test case:

  ```
  write(f, "X");
  ```
  (51)

  Finally, the following sentences in the APL will be added to *CL* for the current test case:

  ```
  close(f);
  ```
  (52)

3.2.14 FTCRL's String Type

One or more specification variables can be transformed in a ⟨*ZExprString*⟩ (see Section 2.8). The result of this transformation is a string-typed expression[6], i.e. it simply is a character string loosing its Z type. From the FTCRL perspective, it only make sense to refine a string-typed expression, *S*, as described by any of the following refinement expressions.

Simple Types

- *S* `==> a` where `a` is a variable of type `string`, then the following sentence in the APL will be generated for the current test case:

  ```
  string a = "X";
  ```
  (53)

[6]String-typed should be read as character-string-typed.

References

- *S* `==> a AS REF[b]` where `a` is a variable of type `string ref`, and `b` is of type `string`. Then the following sentence in the APL will be generated for the current test case:

```
type a = &b;
```
(54)

- *S* `==> a AS REF[b]` where `a` and `b` are variables of type `string ref`. Then the following sentence in the APL will be generated for the current test case:

```
type a = b;
```
(55)

- *S* `==> a AS REF[r.b]` where `a` is a variable of type `string ref`, and `r` is some record such that `b` of type `string` is one of its members. Then the following sentence in the APL will be generated for the current test case:

```
type a = &r.b;
```
(56)

- *S* `==> a AS REF[r.b]` where `a` is a variable of type `string ref`, and `r` is some record such that `b` of type `string ref` is one of its members. Then the following sentence in the APL will be generated for the current test case:

```
type a = r.b;
```
(57)

Members of a Record-Type

- *S* `==> r.a AS REF[b]` where `r` is some record such that `a` of type `string ref`, and `b` is of type `string`. Then the following sentence in the APL will be generated for the current test case:

```
r.a = &b;
```
(58)

- *S* `==> r.a AS REF[b]` where `r` is some record such that `a` of type `string ref`, and `b` is of type `string ref`. Then the following sentence in the APL will be generated for the current test case:

```
r.a = b;
```
(59)

- *S* `==> r.a` where `r` is some record such that `a` of type `string` is one of its members. Then the following sentence in the APL will be generated for the current test case:

```
r.a = "X";
```
(60)

Columns of Tables and Files

- *S* `==> t.a` where `t` is some table such that `a` of type `string`, is one of its columns. Then the following sentence in the APL will be added to *AL* for the current test case:

```
insert(c, t, ..., "a = "X"", ...)
```
(61)

  By enclosing `X` in double quotes we mean that `X` is passed as a character string.

  Besides, the following sentence will be added to *DL*:

```
delete(c, t);
```

  Same considerations of (14) apply for the rest of the elements of these sentences.

- S `==> a AS FILE[...]` where `a` is a file name. Then the following sentence in the APL will be added to DL for the current test case:

```
f = open(a);
```
(62)

in this way the file is opened and emptied at the beginning of the test case. Besides, the following sentence in the APL will be added to AL for the current test case:

```
write(f, "X");
```
(63)

Finally, the following sentence in the APL will be added to CL for the current test case:

```
close(f);
```
(64)

3.2.15 Cross Products

From the FTCRL perspective, it only make sense to refine an expression whose type is a cross product $X_1 \times \cdots \times X_n$ as follows.

- As a record `r` of type `name` whose fields are $name_1, \ldots, name_n$. That is the rule is of the form:

$$\begin{array}{l}\texttt{X ==> r AS RECORD}\\ \quad\texttt{WITH [} X_{i_1} \texttt{ ==> name.}name_{i_1} \texttt{ ...;}\\ \qquad\ldots\\ \qquad X_{i_k} \texttt{ ==> name.}name_{i_k} \texttt{ ...]}\end{array}$$
(65)

where $i_1, \ldots i_k$ is a subset of $1..n$. In this case the k refinement sentences inside the `WITH` directive are processed before this law yielding a variable, `r1` of type `name` whose fields $name_{i_1}, \ldots, name_{i_k}$ have been assigned. Then the following sentence in the APL will be added to AL for the current test case:

```
r = r1;
```
(66)

If the `WITH` clause is absent then proceed as the refinement sentence would has been the following:

$$\begin{array}{l}\texttt{X ==> r AS RECORD}\\ \quad\texttt{WITH [} X_1 \texttt{ ==> name.}name_1 \texttt{ ...;}\\ \qquad\ldots\\ \qquad X_n \texttt{ ==> name.}name_n \texttt{ ...]}\end{array}$$
(67)

where the fields of `r` are ordered as in the definition of the type. Note that in this case a very simple refinement is assumed.

- As a table `t` whose columns are $name_1, \ldots, name_n$ as described in file `f`. That is the rule is of the form:

$$\begin{array}{l}\texttt{X ==> r AS TABLE[c, p, f]}\\ \quad\texttt{WITH [} X_{i_1} \texttt{ ==> f.}name_{i_1} \texttt{ ...;}\\ \qquad\ldots\\ \qquad X_{i_k} \texttt{ ==> f.}name_{i_k} \texttt{ ...]}\end{array}$$
(68)

where $i_1, \ldots i_k$ is a subset of $1 \ldots n$. In this case the k refinement sentences inside the `WITH` directive are processed before this law inserting values in `t` according to the rules given in previous sections. Nothing needs to be done in this case.

If the `WITH` clause is absent then proceed as the refinement sentence would has been the following:

$$\begin{array}{l}\texttt{X ==> r AS TABLE[c, p, f]}\\ \qquad\texttt{WITH [} X_1 \texttt{ ==> f.}name_1 \texttt{ ...;}\\ \qquad\qquad \ldots\\ \qquad\qquad X_n \texttt{ ==> f.}name_n \texttt{ ...]}\end{array} \tag{69}$$

where the fields of `t` are ordered as in file `f`. Note that in this case a very simple refinement is assumed.

- As a file `a`. That is the rule is of the form:

$$\begin{array}{l}\texttt{X ==> a AS FILE[...]}\\ \qquad\texttt{WITH [} X_{i_1} \texttt{ ==> ...;}\\ \qquad\qquad \ldots\\ \qquad\qquad X_{i_k} \texttt{ ==> ...]}\end{array} \tag{70}$$

where $i_1, \ldots i_k$ is a subset of $1 \ldots n$. In this case the k refinement sentences inside the `WITH` directive are processed before this law yielding k initialized variables named `a_1, ..., a_k`. Then the following sentence in the APL will be added to *DL* for the current test case:

```
f = open(a);
```
(71)

in this way the file is opened and emptied at the beginning of the test case. Besides, the following sentences in the APL will be added to *AL* for the current test case:

```
write(f, str(a_1));
...
write(f, str(a_k));
```
(72)

where `str()` is an underspecified function that tries to convert any type into a string. That is, FTCRL will try to serialize each variable and will write the result in the file. Maybe the most convenient way to ensure that these values are saved as expected is to make the left hand side of the refinement sentences inside the `WITH` directive to be FTCRL string expressions, and the right hand sides string variables, but other alternatives are available.

If the `WHILE` directive is absent but all the X_i are either $\mathbb{Z}$, a basic type or an enumerated type, then FTCRL will convert each component into a string and will save them in the file.

Finally, the following sentence in the APL will be added to *CL* for the current test case:

```
close(f);
```
(73)

3.2.16 Sets

When sets are refined, FTCRL imposes no particular order to their elements. However, when a refinement law has two or more expressions involving `.@DOM`, `.@RAN` or *.digit* applied to the same specification variable, then the variable is considered a list (in any arbitrary order), then the expressions are evaluated and their results are assumed to be lists. In this way, when each refinement sentence is processed the components of the original elements belonging to the set, are processed in the same order. For example, if $\{x_1 \mapsto y_1, \ldots, x_m \mapsto y_m\}$ is the value of some specification variable and we have a refinement law such as:

```
f ==> s AS LIST WITH[f.@DOM ==> sdata.x; f.@RAN ==> sdata.y]
```

then `s` will be $\langle [x = x_1, y = y_1], \ldots, [x = x_m, y = y_m] \rangle$. That is, the components of a given ordered pair in f go to the same node of the list.

Consider a refinement rule which contains two laws, r_1 and r_2, whose left hand sides are sets. The right hand side of r_1 has a `REF` clause whose variable points to the implementation variable of r_2. As was explained earlier, r_2 is processed before r_1. Further, this refinement rule make sense only if the set used in r_1 is a subset of the set used in r_2. For example, say we have $f : X \pfun Y$ and $g : \mathbb{P}\, X$. Assume the implementation for g is a list of some type, and that for f is another list of records such as one of its fields is a reference to the list implementing g. Then, this is consistent only if $\mathrm{dom}\, f \subseteq g$, because otherwise there will be some element of f pointing to nowhere. So, when each element in the set of r_1 is processed, FTCRL will set the reference to the corresponding element in the other set. For example, in the case of f and g and assuming $g = \{x_1, x_2, x_3\}$ and $f = \{x_2 \mapsto y_2, x_1 \mapsto y_1\}$, when $x_2 \mapsto y_2$ is considered, the reference for this ordered pair will point to the node where x_1 was stored in the implementation of g. In other words, FTCRL will look up x_1 in the implementation of g, will take the address of that variable and will use this value to set the reference in the implementation of f.

Sets whose elements belong to a Z type. From the FTCRL perspective, it only make sense to refine an expression $\{x_1, \ldots, x_m\}$ whose type is $\mathbb{P}\, X$, as described by any of the following refinement expressions.

- $\{x_1, \ldots, x_m\}$ `==>` $\langle$*regExpr*$\rangle$ see Section 3.2.8.
- $\{x_1, \ldots, x_m\}$ `==> a AS LIST` where `a` is of type `list[type]` and `type` is any type such that X can be refined to it as described by any other semantics rule. The first sentence is:

  ```
  a = mem(type,m);
  ```
  (74)

  And then, apply that rule to each and every element of the set using `a[`i`]` as the target variable for x_i.
- $\{x_1, \ldots, x_m\}$ `==> a AS LIST[SLL,n] WITH[`$\langle$*refinement*$\rangle${`;`$\langle$*refinement*$\rangle$}`]` where `a` is of type `ref type` where `type` is any APL record-type and `n` of type `ref type` is one of its members. As was explained earlier the refinement specification inside the `WITH` directive has already been processed, producing m new implementation variables named `a_1, ..., a_m` each of which is initialized. Then the following sentences in the APL will be generated for the current test case:

  ```
  a = &a_1;
  a_1.n = &a_2;
  ..........;
  a_m-1.n = &a_m;
  a_m.n = NULL;
  ```
  (75)
- $\{x_1, \ldots, x_m\}$ `==> a AS LIST[DLL,n,p] WITH[`$\langle$*refinement*$\rangle${`;`$\langle$*refinement*$\rangle$}`]` where `a` is of type `ref type` where `type` is any APL record-type and `n` and `p` both of type `ref type` are two of its members. As was explained earlier the refinement specification inside the `WITH` directive has already been processed, producing m new implementation variables named `a_1, ..., a_m`

each of which is initialized. Then the following sentences in the APL will be generated for the current test case:

```
a = &a_1;
a_1.p = NULL;
a_1.n = &a_2;
a_2.p = &a_1;
a_2.n = &a_3;
..........;
a_m-1.p = &a_m-2;
a_m-1.n = &a_m;
a_m.p = &a_m-1;
a_m.n = NULL;
```
(76)

- $\{x_1, \ldots, x_m\}$ `==> a AS LIST[CLL,n] WITH[`⟨*refinement*⟩`{;`⟨*refinement*⟩`}]` where `a` is of type `ref type` where `type` is any APL record-type and `n` of type `ref type` is one of its members. As was explained earlier the refinement specification inside the `WITH` directive has already been processed, producing m new implementation variables named `a_1, ..., a_m` each of which is initialized. Then the following sentences in the APL will be generated for the current test case:

```
a = &a_1;
a_1.n = &a_2;
..........;
a_m-1.n = &a_m;
a_m.n = &a_1;
```
(77)

- $\{x_1, \ldots, x_m\}$ `==> a AS LIST[DCLL,n,p] WITH[`⟨*refinement*⟩`{;`⟨*refinement*⟩`}]` where `a` is of type `ref type` where `type` is any APL record-type and `n` and `p` both of type `ref type` are two of its members. As was explained earlier the refinement specification inside the `WITH` directive has already been processed, producing m new implementation variables named `a_1, ..., a_m` each of which is initialized. Then the following sentences in the APL will be generated for the current test case:

```
a = &a_1;
a_1.p = NULL;
a_1.n = &a_2;
a_2.p = &a_1;
a_2.n = &a_3;
..........;
a_m-1.p = &a_m-2;
a_m-1.n = &a_m;
a_m.p = &a_m-1;
a_m.n = &a_1;
```
(78)

- $\{x_1, \ldots, x_m\}$ `==> a AS ARRAY` where `a` is an array of `type` with at least n components such that X can be refined to `type` it as described by any other semantics rule.Then, apply that rule to each and every element of the set using `a[`i`]` as the target variable for x_i.

- $\{x_1, \ldots, x_m\}$ `==> a AS ARRAY WITH[`⟨*refinement*⟩`{;`⟨*refinement*⟩`}]` where `a` is an array of type `type` with at least n components. As was explained earlier the refinement specification inside the `WITH` directive has already been processed, producing m new implementation variables

named `a_1, ..., a_m` each of which is initialized. Then the following sentences in the APL will be generated for the current test case:

```
a[1] = a_1;
.........;
a[m] = a_m;
```
(79)

- $\{x_1, \ldots, x_m\}$ `==> a AS MAPPING WITH[`$\langle$*refinement*$\rangle${`;`$\langle$*refinement*$\rangle$}`]` where `a` is a standard data structure that stores key-value pairs. It is assumed that this this data structure stores this ordered pairs by means of a function named `put()` which takes three arguments: the first one is an instance of the data structure itself, the second one is of the type of the keys and the third one of the type of the values. As was explained earlier the refinement specification inside the `WITH` directive has already been processed, producing $2 * m$ new implementation variables named `k_1, ..., k_m, v_1, ..., v_m` each of which is initialized. Then the following sentences in the APL will be generated for the current test case:

```
put(a, k_1, v_1);
.........;
put(a, k_m, v_m);
```
(80)

- $\{x_1, \ldots, x_m\}$ `==> a AS FILE[...]` where `a` is a file. Same as with cross products and files but treating each element of the set as a component of the cross product are treated.

- $\{x_1, \ldots, x_m\}$ `==> t.a` where `t` is some table such that `a` of type `string`, `float`, `int`, `char` or `enum` is one of its columns. Then apply the rule corresponding to the type of the set and the type of `a` for each element of the set.

- $\{x_1, \ldots, x_m\}$ `==> t AS TABLE[...]  WITH[...]` where `t` is some table. Then, first process the refinement sentences inside the `WITH` directive. These should be of the form described in the previous rule. Nothing else needs to be done.

Sets whose type is FTCRL's String Type. This is the result of one or more sets participating in a $\langle$*zExprString*$\rangle$ expression, without applying the `STR` modifier to them. If two ore more sets participate in the expression, then apply the considerations described at the beginning of this section. Then the refinement expression is of the form:

$[\langle$*zExprString*$\rangle$ `++` $]\{x_1, \ldots, x_m\}[$ `++` $\langle$*zExprString*$\rangle]$ `==>` $\langle$*refinement*$\rangle$

Then, the refinement expression is converted into the following set:

$\{[\langle$*zExprString*$\rangle$ `++` $]x_1[$ `++` $\langle$*zExprString*$\rangle], \ldots, [\langle$*zExprString*$\rangle$ `++` $]x_m[$ `++` $\langle$*zExprString*$\rangle]\}$

and this set is processed as a set whose elements belong to a Z type (that is, all the previous rules are applied).

Sets whose elements belong to an implementation type. This situation arises when the left hand side of a refinement sentence is a call to an implementation function and one of its arguments is of a set type. If two ore more sets participate in the expression, then apply the considerations described at the beginning of this section.

Consider an expression such as:

```
f({x_1, ..., x_m} ==> y) ==> w ...
```

then transform it as follows:

```
{f(x_1 ==> y), ..., f(x_m ==> y)} ==> w ...
```

The type of the last set is $\mathbb{P}$ `X` where `X` is the (implementation) type returned by `f`. Since the type of the last left hand expression is some set, then from the FTCRL perspective it only make sense to refine it as described above in this section, except for mappings. Then, in general, apply the same rules with the following exceptions.

- The implementation variable (`w`) is an array and the type of the set is the same of the indexes of the array; or, the implementation variable is a list and the type of the set is `int`. In this case, if the value returned by the function, for some combination of actual arguments, is n then whatever has to be stored in the list or array must go in position n. In general, the `WITH` directive will say what to store in the list or array, but if the list or array stores simple types, then no `WITH` directive is necessary and FTCRL stores the first parameter passed to the function call whose type is compatible, according to the typing rules of the target programming language, with the type of the list or array.
- In any other case. Each element returned by the function is stored in the list or array as they appear. In this case, the type of the implementation variable must be compatible, according to the typing rules of the target programming language, with the type returned by the function.

3.2.17 Sequences

Although in Z sequences are sets of ordered pairs and, thus, they should be treated according to the semantics rules of section 3.2.16, FTCRL treats them slightly different. Semantics rules for refining a sequence are exactly the same than those for refining sets except that its elements are refined in strict order with respect to the list.

3.2.18 Schema Types

Since Fastest currently does not fully support schema types we do not go into details on this. However, the semantic rules for schema types are essentially the same of those for cross products.

3.2.19 The `@UUT` and `@EPILOGUE` Sections

Once the interpreter has processed the `@LAWS` section, it generates the code corresponding to the `@UUT` section as follows:

- `@UUT f(p_1, ..., p_n)` produces the following APL sentence:

```
f(p_1, ..., p_n);
```
(81)

- `@UUT f(p_1, ..., p_n) MODULE m` produces the following APL sentence:

```
m.f(p_1, ..., p_n);
```
(82)

Finally, the interpreter blindly copies the contents of the `@EPILOGUE` section as did with the `@PREAMBLE`.

3.2.20 The Concrete Test Case

The concrete test case corresponding to the current test case is produced as follows:

$$
\begin{aligned}
&\mathit{contents}(\texttt{@PREAMBLE}) \\
&\quad \frown (1) \\
&\quad \frown \mathit{DL} \frown \mathit{AL} \frown \mathit{CL} \frown \mathit{contents}(\texttt{@PLCODE}) \frown ((81) \mid (82)) \frown \mathit{contents}(\texttt{@EPILOGUE}) \frown \texttt{end} \qquad (83)
\end{aligned}
$$

where $\frown$ means string concatenation.

4 User Commands

The FTCRL interpreter is invoked from within Fastest. We call this process *test case refinement*. Before invoking it the user should have generated some abstract test cases for some operations of a given specification. The first step in test case refinement is to load (into Fastest) at least one refinement rule—for now, it is assumed that refinement rules are written with any text editor and saved in a text file; the standard extension for these files might be `.tcrl`. The command to do this is as follows:

```
loadrefrule path
```

where `path` is the path to a file containing a refinement rule. This command parses and checks the refinement rule informing syntax errors. Note that the file name might be different from the rule name introduced with `@RRULE`.

Once a refinement rule has been loaded the user can refine one or more abstract test cases. The command to do this is as follows:

```
refine <name> to test <path> implemented in <pl> with <frr>
```

where:

- `<name>` is the name of either a Z operation (cf. Fastest's `selop` command) or an abstract test case. In the first case all the abstract test cases of the operation are refined, in the second only that test case is refined.
- `<path>` is a path to a directory where the source files containing the code of the UUT referenced in `<frr>` are saved.
- `<pl>` is the name of the programming language to which test cases are going to be refined. Possible values are: `C` and `Java`. After the name of the programming language it is possible to pass a list of arguments as follows:

  ```
  -arg=val -arg=val ... -arg=val
  ```

  If some argument is not given the interpreter assumes a default value. The arguments currently supported by Fastest are listed in Table 1. Each of these arguments captures some technological issue on which refinement may depend on. These dependencies are described in the semantic rules of each programming language.

PL	`database`	`memadmin`	`os`	`filesys`
Desc.	Database technology: connections, SQL, etc.	API for memory administration; used to allocate memory for dynamic data structures	Module managing operating system dependencies; it is the operating system on which the tests are going to be executed.	Module managing file system dependencies
C	`sqlite`	`libc`	`linux`, `windows`	
Java	`jdbc`		`linux`, `windows`	

Table 1: Arguments to `<pl>`. Each cell lists the possible values, the default value is boxed.

- `<frr>` is the name of a refinement rule (cf. `@RRULE`) previously loaded with `loadrefrule`.

The refinement command saves the list of concrete test cases in memory. Users can see and export them to files with the following command:

```
showctc name | -all [-o path]
```

where `name` is the name of a concrete test case and `path` is a path to some directory. The name of a concrete test case is equal to its corresponding abstract test case except that `TCASE` is replaced by `CTCASE`. If `name` is passed only that concrete test case is displayed; if `-all` is passed, all the concrete test cases are displayed. If the optional argument `-o` is passed then Fastest saves the concrete test case or all, depending of the first argument, in a file stored in `path` instead of displaying them on screen. The names of these files are the names of the concrete test cases with the default file name extension of the `<pl>` argument passed to `refine`.

5 Semantics for the C Programming Language

The FTCRL semantics when the target programming language is ANSI C will be described by:

1. Giving the C types FTCRL-equivalent to the APL types.
2. Writing C code for each semantic rule (1)-(83). If one of these equations is omitted it means that the code is exactly the same.

5.1 Type Equivalences

Table 2 shows the equivalences between APL types and C types. We say that an APL type is FTCRL-equivalent to a C type if they are listed in the same row of Table 2. If some C type is FTCRL-equivalent to some APL type it means that the semantic rules that are applicable to the last must be applied to the former. For instance APL's `int` is FTCRL-equivalent to C's `long int`, then the semantic rules that are applicable to `int` must be applied to `long int`. When a semantic rule is applied to a FTCRL-equivalent type, then the type in the declarations must be changed accordingly. For example, when semantic rule (2) is applied to a `long int`, then it becomes:

```
long int a = X;
```

Besides, the syntax of declarations might slightly change. For example, in APL we write:

```
colors enum [red, white, blue];
```

but in C we write:

```
enum colors {red, white, blue};
```

However, if these are the only differences between an APL semantic rule and the corresponding C rule, we will not include it in this section assuming that the C rule is identical to the APL rule modulo the changes we just mentioned.

APL type	**C types**
`int`	`int`, `short int`, `short`, `long int`, `long`, `unsigned int`, `unsigned short int`, `unsigned short`, `unsigned long int`, `unsigned long`, `signed int`, `signed short int`, `signed short`, `signed long int`, `signed long`
`char`	`char`, `unsigned char`, `signed char`
`float`	`float`, `double`, `long double`
`enum`	`enum`
Arrays	The same
`record`	`struct`
`list[type]`	`type *`
`string`	`char *`
`type ref`	`type *`

Table 2: FTCRL-equivalent type for the C programming language.

The ambiguity of the last three rows of Table 2 justifies the inclusion of the `AS` directive in FTCRL.

5.2 Semantic Rules

In this section we list those semantic rules introduced in Section 3 that must be changed when the target programming language is C. Some have been omitted as explained in the previous sections. Same considerations than those introduced in sections 3.2.2, 3.2.3 and 3.2.4 apply.

All of the sentences involving system calls could be enclosed inside conditional sentences capturing possible errors. If an error is detected the concrete test case shall be abandoned.

(C1) (1) becomes

```
int main() {
if (!init()) {print("Initialization error"); return 0;}
```

(CChar) Elements of `char` and its FTCRL-equivalents are written enclosed in simple quotes.

(C14) (14) depends on the `database` parameter passed to the interpreter when it is called (see Section 4). Therefore, we consider each of the possible values of this argument:

- `sqlite`. The following is added to *DL*:

  ```
  sqlite3_open_v2 (t, &c, SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE, NULL);
  sqlite3_exec (db, "TRUNCATE t;", NULL,  NULL,  NULL);
  ```

  The following code is added to *AL*:

  ```
  sqlite3_exec (db,
                "INSERT INTO t (..., a, ...)
                        VALUES (..., XX, ....);", NULL,  NULL,  NULL);
  ```

(C15) In (15)-(17) the sentences must be adapted to the standar library, but they are very similar.

(C27) (31) same as (C14).

(C30-0) In (32)-(34) the sentences must be adapted to the standar library, but they are very similar.

(C30) (37) becomes:

```
int a = k + m;
```

(C32) In (39) $k + m$ is converted into character and not $k + m - 1$.

(C41) In (48) consider $k + m$ and not $k + m - 1$.

(C42) (49) same as (C14).

(C44) (62)-(64) depends on the `os` parameter passed to the interpreter when it is called (see Section 4). However, most of the code is independent of this parameter.

The following is added to *DL*:

```
fd = open(pathf, O_RDWR | O_TRUNC | O_CREAT);
```

where `pathf` is defined below.

The following code is added to *AL*:

```
write(fd, "X", strlen("X"));
```

Finally, when all the refinement laws have been processed the interpreter shall append the following code to AL:

```
close(fd);
```

The code that depends on the `os` parameter is as follows:

- `linux`: `pathf` equals the following string expression `path ++ "/" ++ a`
- `windows`: `pathf` equals the following string expression `"C:\\" ++ pathw ++ "\\" ++ a` where `pathw` equals `path` but all the `/` characters are replaced by `\\`.

(C46) In (50)-(52) the sentences must be adapted to the standar library, but they are very similar.

(C52) (61) same as (C14).

(C53) In (74) the sentence `mem(type,m)` depends on the `memadmin` parameter passed to the interpreter when it is called (see Section 4). Therefore, we consider each of the possible values of this argument:

- `libc`: `mem(type,m)` becomes `(type *) calloc(sizeof(type),m)`

Regardless of this parameter, each of the sentences of the form `a[i] = a_i` becomes:

```
*(a + i) = a_i;
```

and `i` starts at zero.

(C57) In (62)-(64) the sentences must be adapted to the standar library, but they are very similar.

(C60) (81) must be a C function whose name is different from `main()`.

(C61) (82) is ignored.

5.3 The Concrete Test Case

The concrete test case corresponding to the current test case is produced as follows:

$$contents(\texttt{@PREAMBLE}) \frown (1) \frown DL \frown AL \frown CL \\ \frown contents(\texttt{@PLCODE}) \frown (C60) \frown contents(\texttt{@EPILOGUE}) \frown \texttt{return 1;}\}$$

6 Semantics for the Java Programming Language

The FTCRL semantics when the target programming language is Java ***PONER LA VERSION*** will be described by:

1. Giving the Java types FTCRL-equivalent to the APL types.
2. Writing Java code for each semantic rule (1)-(83). If one of these equations is omitted it means that the code is exactly the same.

6.1 Type Equivalences

APL type	Java types
`int`	`int`, `short`, `long`, `byte`, `Integer`, `Short`, `Long`, `Byte`
`char`	`char`, `Character`
`float`	`float`, `double`, `Float`, `Double`
`enum`	`enum`
Arrays	The same
`record`	`class`
`list[type]`	`List<type>`, `ArrayList<type>`, `LinkedList<type>`
Mappings	`Attributes`, `HasMap`, `Hashtable`, `IdentityHashMap`, `TreeMap`, `WeakHashMap`
`string`	`String`
`type ref`	All variable declarations actually declare a reference, except for basic types in which case it is impossible to declare a reference

Table 3: FTCRL-equivalent type for the Java programming language.

Table 3 shows the equivalences between APL types and Java types. We say that an APL type is FTCRL-equivalent to a Java type if they are listed in the same row of Table 3. If some Java type is FTCRL-equivalent to some APL type it means that the semantic rules that are applicable to the last must be applied to the former. For instance APL's `int` is FTCRL-equivalent to Java's `long`, then the semantic rules that are applicable to `int` must be applied to `long`. When a semantic rule is applied to a FTCRL-equivalent type, then the type in the declarations must be changed accordingly. For example, when semantic rule (2) is applied to a `long`, then it becomes:

```
long a = X;
```

Besides, the syntax of declarations might slightly change. For example, in APL we write:

```
colors enum [red, white, blue];
```

but in Java we write:

```
enum colors {red, white, blue};
```

However, if these are the only differences between an APL semantic rule and the corresponding Java rule, we will not include it in this section assuming that the Java rule is identical to the APL rule modulo the changes we just mentioned.

6.2 Semantic Rules

In this section we list those semantic rules introduced in Section 3 that must be changed when the target programming language is Java. Some have been omitted as explained in the previous sections. Same considerations than those introduced in sections 3.2.2, 3.2.3 and 3.2.4 apply.

All of the sentences involving system calls could be enclosed inside `try-catch` structures capturing possible exceptions. If an exception is detected the concrete test case shall be abandoned.

The `Init` Class. As was explained in sections 2.2 and 3.2.3, the preamble must contain the definition of a function called `init()`. Since in Java functions can exist only as class methods, the `init()` function is assumed to be a method of a class called `Init`. Then, this class must be defined in the preamble by the tester. This class must export at least a method called `init()` which receives no parameters and returns an `int`.

Accessing Global Resources. Sometimes a refinement rule refers to some resources that are assumed to be accessible from each concrete test cases. This is the case, for instance, of an object that represents the connection to access a database (see section 2.9.7) or an implementation function that is called from a refinement law. All of these resources are assumed to be declared and initialized inside the preamble. Moreover, it is assumed that all of them are public entities of the class `Init`. In other words, if some refinement rule declares a law such as:

```
tbl1:clients ==> clientData AS TABLE[c, /home/usr, clientData.txt] WITH[...]
```

then the `Init` class must make `c` a public object accessible as follows:

```
init.c
```

where `init` is some instance of `Init`.

The same is valid for implementation functions called from a refinement law. For example, if we have:

```
A ==> f(A ==> x) ==> xs AS LIST...
```

then `f` must be declared as a public method of `Init` and so it will be used as follows:

```
init.f(...)
```

Obviously, `Init.f()` will just call the true function declared in some other place.

(J1) (1) becomes:

```
public class main {
  public static void main(String []) {
     Init init = new Init();
     if (!init.init())
       {System.out.print("Initialization error"); System.exit(0);}
     m test = new m();
```

where `m` is the argument to the `MODULE` directive of the `@UUT` section, i.e. is the class under test, i.e. one of the methods of `m` will be tested.

Very Important – Reflection. When the target programming language is Java, Fastest will always test a method of some class. Therefore, the implementation variables in a refinement rule are either parameters of the method or class fields. If a class field is declared as private, then it cannot be initialized from outside the class unless there are some method to do that [AGH00]. Since the existence of such methods is not guaranteed we decided to rest on reflection to initialize private variables. Hence, whenever one of the semantic rules has to be applied to a private variable of the class being tested, the initialization part must be implemented with reflection. The code to do this is divided in two parts:

1. For each private variable, `var`, appearing in the refinement rule the interpreter will include in *DL* the following code:

   ```
   Field var = test.getClass().getDeclaredField("var");
   var.setAccessible(true);
   ```

   to make the variable accessible from the outside of the class by means of reflection. `test` is an object declared in (J1).

2. The initialization of each private variable, `var`, whose type is not a primitive type, to be included in *AL* is then:

   ```
   var.set(test, X);
   ```

   where `X` is the value to assign to `var` according to the corresponding semantic rule. When `var` is of a primitive type `type` the code must be:

   ```
   a.setType(test, X);
   ```

However, we will keep semantic rules as if they were for public variables, that is as if they could be initialized without reflection. For instance, in (J2) we said the code is:

```
type a = new type(X);
```

for some types, but if `a` is a private variable then the code must be:

```
Field a = test.getClass().getField("a");
```

in *DL* and

```
a.set(test, X);
```

in *AL*.

(J2) (2) changes only when the implementation type, `type`, is a non primitive, FTCRL-equivalent to APL's `int`. In these cases the code is:

```
type a = new type(X);
```

(J3) (3) changes only when the implementation type, `type`, is a non primitive, FTCRL-equivalent to APL's `float`. In these cases the code is:

```
type a = new type(X);
```

(JChar) Elements of type `char` are written enclosed in simple quotes.

(J4) (4) changes only when the implementation type is `Character`. In this case the code is:

```
Character a = new Character('cX');
```

(J5) (5) changes to:

```
name a = name.constX;
```

(J7) (7) are (8) are valid only when `a` and `b` are of any non primitive Java type FTCRL-equivalent to one of the APL types mentioned in the rule. The code is:

```
type a = b;
```

(J9) (9) and (10) are valid only when `a` and `b` are of any non primitive Java type FTCRL-equivalent to one of the APL types mentioned in the rule. The code is:

```
type a = r.b;
```

(J11) (11) and (12) are valid only when `a` and `b` are of any non primitive Java type FTCRL-equivalent to one of the APL types mentioned in the rule. The code is:

```
type a = r.b;
```

(J14) (14) depends on the `database` parameter passed to the interpreter when it is called (see Section 4). Therefore, we consider each of the possible values of this argument:

- `jdbc`. The following is added to *DL*:

  ```
  Statement stmt = init.c.createStatement();
  stmt.executeUpdate("delete " + t);
  ```

  The following code is added to *AL*:

  ```
  stmt.executeUpdate("insert into " + t + "values(..., XX, ....)");
  stmt.close();
  ```

  where the values are ordered according to the table definition.

(J15) (18) changes only when the implementation type, `type`, is a non primitive, FTCRL-equivalent to APL's `int`. In these cases the code is:

```
type a = new type(iX);
```

(J16) (19) changes only when the implementation type is, `type`, a non primitive, FTCRL-equivalent to APL's `float`. In these cases the code is:

```
type a = new type(iX);
```

(J17) (20) changes only when the implementation type is `Character`. In this case the code is:

```
Character a = new Character('ciX');
```

(J18) (21) changes to:

```
name a = name.constiX;
```

(J20) same as (J7)

(J22) same as (J9)

(J24) same as (J11)

(J27) same as (J14)

(J28) (35) changes to:

```
name a = name.constK;
```

(J29) (36) changes to:

```
name a = name.constJK;
```

(J30) (37) changes to

```
type a = k + m;
```

when `type` is a primitive type FTCRL-equivalent to `int`. When `type` is a non primitive, FTCRL-equivalent to APL's `int` type, then the code is:

```
type a = new type(k + m);
```

(J31) (38) changes only when the implementation type, `type`, is a non primitive, FTCRL-equivalent to APL's `int`. In these cases the code is:

```
type a = new type(Jk);
```

(J32) In (39) $k + m$ is converted into a character and not $k + m - 1$. Besides, the code changes to:

```
Character a = new Character('ckX');
```

when the implementation type is `Character`.

(J33) (40) changes to:

```
Character a = new Character('cjkX');
```

when the implementation type is `Character`.

(J34) (41) changes to:

```
String a = new String("ConstK");
```

(J35) same as (J7)

(J37) same as (J9)

(J39) same as (J11)

(J42) same as (J14)

(J43) (53) changes to:

```
String a = new String("X");
```

(J44) (62)-(64) changes as follows. Add this to *DL*:

```
File file = new File("pathf");
writer = new BufferedWriter(new FileWriter(file));
```

where `pathf` equals the following string expression `path ++ "/" ++ a`.

The following code is added to *AL*:

```
writer.write("X");
```

Finally, when all the refinement laws have been processed the interpreter shall append the following code to *AL*:

```
writer.close();
```

(J45) same as (J7)

(J47) same as (J9)

(J49) same as (J11)

(J52) same as (J14)

(J53) (74) is valid only when the implementation type is `List` or `ArrayList`. It becomes:

`type a_1, ..., a_m;`
refine(x_1, `a_1`)
.....
refine(x_m, `a_m`)
`a.add(a_1);`
`.....;`
`a.add(a_m);`

(J54) same as (J53)

(J55) (76) is valid only when the implementation type is `LinkedList`. It becomes:

`type a_1, ..., a_m;`
refine(x_1, `a_1`)
. . . . . .
refine(x_m, `a_m`)
`a.add(a_1);`
`.....;`
`a.add(a_m);`

(J56) same as (J53). It is assumed that the implementation type provides the illusion of a circular list by using a convenient iterator.

(J57) same as (J55). It is assumed that the implementation type provides the illusion of a circular list by using a convenient iterator.

(JMap) (80) is valid only when `a` is of any of the types listed in row "Mappings" in Table 3. It becomes:

`K k_1, ..., k_m;`
`V v_1, ..., v_m;`
refine(*withRefinementK*(x_1), `k_1`)
. . . . . .
refine(*withRefinementK*(x_m), `k_m`)
refine(*withRefinementV*(x_1), `v_1`)
. . . . . .
refine(*withRefinementV*(x_m), `v_m`)
`a.put(k_1, v_1);`
`.....;`
`a.put(k_m, v_m);`

(J60) (81) is ignored

(J61) (82) is the only way to test Java units. In Java units are always class methods. The code is:

`test.f(p_1, ..., p_n);`

where `test` is a variable of type `m` created in (J1). `f` must be different from `main`.

6.3 The Concrete Test Case

The concrete test case corresponding to the current test case is produced as follows:

contents(`@PREAMBLE`)
$\frown$(1)
$\frown DL \frown AL \frown CL$
$\frown$*contents*(`@PLCODE`) $\frown$ ($J61$) $\frown$ *contents*(`@EPILOGUE`) $\frown$ `System.exit(1);}}`

A FTCRL Grammar

- Strings in `typewriter type` are terminal strings.
- *eol* means "end of line".

⟨*refinementLaw*⟩ ::= `@RRULE` ⟨*name*⟩ *eol*
⟨*preamble*⟩ *eol*
⟨*laws*⟩ *eol*
[⟨*plcode*⟩ *eol*]
⟨*uut*⟩ *eol*
[⟨*epilogue*⟩ *eol*]

⟨*preamble*⟩ ::= `@PREAMBLE` *eol*
⟨*PLCode*⟩ | ⟨*name*⟩`.@PREAMBLE` *eol*
{⟨*PLCode*⟩ | ⟨*name*⟩`.@PREAMBLE` *eol*}

⟨*plcode*⟩ ::= `@PLCODE` *eol*
⟨*PLCode*⟩ *eol*

⟨*uut*⟩ ::= `@UUT` ⟨*iName*⟩`(`[⟨*iName*⟩{`,`⟨*iName*⟩}]`)` [`MODULE` ⟨*iName*⟩] *eol*

⟨*epilogue*⟩ ::= `@EPILOGUE` *eol*
⟨*PLCode*⟩ | ⟨*name*⟩`.@EPILOGUE` *eol*
{⟨*PLCode*⟩ | ⟨*name*⟩`.@EPILOGUE` *eol*}

⟨*laws*⟩ ::= `@LAWS` *eol*
⟨*law*⟩ *eol* | ⟨*reference*⟩ *eol* | ⟨*name*⟩`.@LAWS` *eol*
{⟨*law*⟩ *eol* | ⟨*reference*⟩ *eol* | ⟨*name*⟩`.@LAWS` *eol*}

⟨*reference*⟩ ::= ⟨*name*⟩`.`⟨*lawName*⟩

⟨*law*⟩ ::= [⟨*name*⟩`:`]⟨*lawSynonym*⟩ | [⟨*name*⟩`:`]⟨*lawRefinement*⟩

⟨*lawSynonym*⟩ ::= ⟨*name*⟩`==`(⟨*asSynonym*⟩ | ⟨*withSynonym*⟩)

⟨*asSynonym*⟩ ::= ⟨*asRefinement*⟩

⟨*withSynonym*⟩ ::= ⟨*withRefinement*⟩

⟨*lawRefinement*⟩ ::= ⟨*sName*⟩ {`,`⟨*sName*⟩} `==>` ⟨*refinement*⟩{`;`⟨*refinement*⟩}

⟨*refinement*⟩ ::= ⟨*iName*⟩[`AS` ⟨*asRefinement*⟩ | ⟨*asSynonym*⟩]
| ⟨*exprRefinement*⟩

⟨*asRefinement*⟩ ::= ⟨*dataStruct*⟩[`WITH[`⟨*withRefinement*⟩`]`]

⟨*withRefinement*⟩ ::= ⟨*exprRefinement*⟩{`,`⟨*exprRefinement*⟩}

⟨*exprRefinement*⟩ ::= ⟨*zExpr*⟩ `==>` ⟨*refinement*⟩

⟨*zExpr*⟩ ::= ⟨*zExprSet*⟩ | ⟨*zExprNum*⟩ | ⟨*zExprString*⟩ | ⟨*zExprSeq*⟩

⟨*zExprSet*⟩ ::= ⟨*sName*⟩[`.`⟨*dotSetOper*⟩] | ⟨*setExtension*⟩ | ⟨*zExprSet*⟩ ∪ ⟨*zExprSet*⟩ | ...

⟨*zExprNum*⟩ ::= ⟨*sName*⟩[`.#`]
| ⟨*number*⟩
| `@AUTOFILL`
| ⟨*zExprNum*⟩ `div` ⟨*zExprNum*⟩
| ⟨*zExprNum*⟩ `/` ⟨*zExprNum*⟩
| ⟨*zExprNum*⟩ `div` ⟨*zExprNum*⟩
| ⟨*zExprNum*⟩ `mod` ⟨*zExprNum*⟩
| ⟨*zExprNum*⟩ `+` ⟨*zExprNum*⟩
| ...

⟨*zExprString*⟩ ::= ⟨*string*⟩
| ⟨*number*⟩
| `@AUTOFILL`
| ⟨*sName*⟩[`.`(⟨*dotSetOper*⟩ | `#` | `@STR`)]
| ⟨*zExprString*⟩ `++` ⟨*zExprString*⟩

⟨*zExprSeq*⟩ ::= ...

⟨*dotSetOper*⟩ ::= `@`(`DOM` | `RAN` | `ELEM`) | ⟨*digit*⟩ | ⟨*sName*⟩ | ⟨*dotSetOper*⟩`.`⟨*dotSetOper*⟩

⟨*dataStruct*⟩ ::= `ARRAY`
| `RECORD` | ⟨*list*⟩ | ⟨*map*⟩ | ⟨*reference*⟩ | ⟨*enumeration*⟩ | ⟨*table*⟩ | ⟨*file*⟩

⟨*list*⟩ ::= `LIST[[`⟨*listType*⟩`,`(⟨*iName*⟩ | ⟨*iName*⟩`,`⟨*iName*⟩)`]]`

⟨*map*⟩ ::= `MAPKEY[`⟨*iName*⟩`,`⟨*iType*⟩[`,`⟨*iName*⟩]`]` | `MAPVAL[`⟨*iName*⟩`,`⟨*iType*⟩`]`

⟨*reference*⟩ ::= `REF[`⟨*iName*⟩`]`

⟨*enumeration*⟩ ::=
`ENUM[[`
(⟨*sName*⟩ `>` (⟨*iName*⟩ | ⟨*number*⟩){`,`⟨*sName*⟩ `>` (⟨*iName*⟩ | ⟨*number*⟩}) | ⟨*number*⟩`]]`

⟨*table*⟩ ::= `TABLE[`⟨*iName*⟩`,`⟨*path*⟩`,`⟨*fName*⟩`]`

⟨*file*⟩ ::= `FILE[`⟨*path*⟩`]`

⟨*listType*⟩ ::= `SLL` | `DLL` | `CLL` | `DCLL`

⟨*PLCode*⟩ ::= legal text of some programming laguage

⟨*name*⟩ ::= ⟨*letter*⟩{_ | ⟨*digit*⟩ | {⟨*name*⟩}}

⟨*sName*⟩ ::= valid Z identifier

⟨*iName*⟩ ::= ⟨*iIdent*⟩ | ⟨*iIdent*⟩`[]` | ⟨*iIdent*⟩`.`⟨*iIdent*⟩ | ⟨*iIdent*⟩`.*`

⟨*iIdent*⟩ ::= valid identifier in the programming language

⟨*fName*⟩ ::= valid identifier of a file in the operating system

⟨*path*⟩ ::= valid path in the Linux operating system

⟨*string*⟩ ::= any charcter string enclosed in double quotes

⟨*setExtension*⟩ ::= any valid Z set extension

⟨*digit*⟩ ::= `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

⟨*number*⟩ ::= ⟨*digit*⟩{`0` | ⟨*digit*⟩}

⟨*letter*⟩ ::= `a` | . . . | `z` | `A` | . . . | `Z`

B TODO List

Here we list some FTCRL features that should be analyzed and perhaps added to it:

- Perhaps it would be necessary that file and table names depend on the values bound to the variables of the current test case. Maybe the best choice is to have something like:

$$\begin{array}{rcl} \langle refinement \rangle ::= & \langle zExprString \rangle \texttt{ AS FILE[...]} & \langle asRefinement \rangle \\ & \langle zExprString \rangle \texttt{ AS TABLE[...]} & \langle asRefinement \rangle \end{array}$$

- Analyze if it is better to separate lines when refining to a file by explicitly adding a `'\n'` character as required by C's `printf()` routine.
- It might be necessary to add support for other external devices such as network connections, serial ports, etc.
- There are some semantic rules for tables that were informally described in Section 2.9.7 but were not formalized in Section 3.
- We need to add support to data structures such as Pascal's variant records or C's unions.
- Maybe it will be necessary to add some form of conditional execution. Something along the lines:

```
l1: f ==> @IF f.# < 10 @THEN a AS ARRAY @ELSE c AS LIST @FI
```

  This might be useful to support the data structures mentioned above.

- It should be investigated whether it is necessary to place the calls to system calls (such as `mem()` in (74) or `insert()` in (14)) within an `if` clause to capture a possible failure and do something:

```
err = insert(c, t, ..., "a = "X"", ...);
if err = 0 then print("insert error"); return 0; endif
```

- Similarly, we should see if the epilogue must be placed within an `if` clause.
- Maybe it would be convenient to introduce a type representing valid dates.
- Study whether is necessary or convenient to allow refinement laws whose left hand side is a list of Z types instead of specification variables. In this way, the interpreter would refine a variable by applying the law for its type.
- Consider removing the `@AUTOFILL` directive and/or adding an `autofill` parameter to the interpreter with a similar semantics. In this way, it is possible to define refinement-rule-wide behaviour for implementation variables not appearing in the refinement rule.

References

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition.* Addison-Wesley, 2000.

[BW98] Ralph-Johan Back and J. Von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language Second Edition.* Prentice-Hall, Inc., 1988.

[Mor94] Carroll Morgan. *Programming from specifications (2nd ed.).* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

TODO:

- Habría que pensar si la notación para arreglos no debería ser de la forma, por ejmplo: `a[f.@DOM] := f.@RAN`. El problema es que solo tiene sentido para arreglos... no se puede armar una lista con acceso directo... En otras palabras habría que ver cómo referir al índice de los arreglos porque se podrían usar solo algunos de ellos o en un orden específico. Tal vez podría ser `a AS ARRAY WITH [f.@RAN ==> a[f.@DOM]]`.

  Esto permite borrar las consideraciones itemizadas en página 51 y 52 de "Sets whose elements belong to an implementation type".

- Hay que permitir acceder a los elementos que componen el valor de retorno de funciones de implementación. Por ejemplo si `A` es un conjunto de la especificación y `f` es una función de la implementación que retorna algo de tipo record o class, entonces un refinamiento de la forma `f(A ==> p) ==>` retorna un conjunto de record o class y cada uno de ellos podría tener que guardarse como una fila en una tabla donde cada columna es uno de los elementos del record o class. Eso debería ir a la derecha del `==>`:

  ```
  f(A ==> p) ==> t AS TABLE[...] WITH[f().f1 ==> t.c1; ...; f().fn ==> t.cn]
  ```

  donde `f1, ..., fn` son los campos del registro o clase que retorna `f`.

- Entre los parámetros de TABLE se pide el directorio y el archivo donde está descripta la tabla. Aparentemente en la implementación se exige que el nombre del archivo sea igual al de la tabla. Si es así no vale la pena poner el nombre del archivo como un parámetro.