Constraint Reasoning in FocalTest

Matthieu Carlier[1], Catherine Dubois[1], and Arnaud Gotlieb[2]

[1] ENSIIE, Évry, FRANCE - {carlier,dubois}@ensiie.fr
[2] INRIA, Rennes, FRANCE - arnaud.gotlieb@irisa.fr

Abstract. Program testing implies selecting test data from the program input space. In many cases, test data satisfying user-specified properties or preconditions (a.k.a. positive test data) are required. However, current automatic test data generation techniques adopt direct *generate-and-test* approaches for this task. In FocalTest, the testing tool of the Focal correct-by-construction environment for developing certified object-oriented functional programs, test data are generated at random and rejected when they do not satisfy preconditions. In this paper, we improve FocalTest with a *test-and-generate* approach, through the usage of constraint reasoning. A particular difficulty is the handling of function calls in the preconditions as they require constraint reasoning on conditionals and pattern matching which introduce disjunctions in constraint systems. Our experimental results show that a non-naive implementation of constraint reasoning on these constructions outperform traditional generation techniques when used to find test data for testing properties.

1 Introduction

The Focal correct-by-construction environment[3] allows one to incrementally build library components with a high level of confidence and quality. A component of a Focal library can contain specifications, implementations of operations and proofs that the implementations satisfy their specifications. Focal components are translated into OCaml executable code and are verified by the Coq proof assistant [2]. The environment Focal also incorporates a testing tool, FocalTest, that allows to automatically test properties. Testing a property or an intermediate lemma can help to prepare a proof by quickly eliminating false conjectures and generate counterexamples to help to correct the lemmas or the implementation they apply to. Furthermore, some of the expected properties of the software under development in the Focal environment will be really proved, others not because they exceed the perimeter of the certified and/or certifiable heart (thus considered as axioms). Preparing a proof by testing is a facility already incorporated into some theorem provers, e.g. Agda [3] and Isabelle [4].

The testing tool we have developed and integrated within Focal, FocalTest shares this approach and is inspired by Quickcheck [5]. Thus, FocalTest allows the user to test implementations against properties from which test data and

[3] Focal [1] is developed by the Focal project (`http://focal.inria.fr`)

oracle can be derived. Properties under test can be decomposed in some preconditions and a conclusion. We are interested in selecting test data satisfying the preconditions of the property under test, such test data are named positive test data. A first version of the tool has been developed where test data are generated at random and rejected when they violate the preconditions [6].

In this paper, we improve FocalTest with a *test-and-generate* approach for test data selection through the usage of constraint reasoning. The solution we propose consists in exploring very carefully the precondition part of the property under test and more precisely the definition of the involved functions in order to produce constraints upon the values of the variables. Then it would remain to instantiate the constraints in order to generate test cases ready to be submitted. The underlying method to extract the constraints is based on the translation of the precondition part of the property under test and the body of the different functions involved in it into an equivalent constraint logical program over finite domains - CLP(FD). Constraint solving relies on domain filtering and constraint propagation, resulting, if any, in solution schemes, that once instantiated will give the expected test cases.

The extraction of constraints and their resolution have required to adapt the techniques developed in [7] to the specification and implementation language of Focal, which is a functional one, close to ML. In particular, an important technical contribution of the paper concerns the introduction in CLP(FD) of constraints related to values of concrete types, i.e. types defined by constructors and pattern-matching expressions.

In this paper we focus on the way to produce and solve constraints obtained from a Focal expression. It is used here to generate positive test data, that is test data that satisfy the precondition part of the property under test. It could also be used to generate test cases that do not valid the precondition (negative test case) or a decision involved in the precondition, as it could be asked to generate e.g. test data that satisfy the MC/DC criterion on the precondition.

The paper is organized as follows. Section 2 proposes a quick tour of the environment Focal and briefly precises the background of our testing environment FocalTest which includes the subset of the language considered for writing programs and properties. Section 3 details our translation scheme of a Focal program into a CLP(FD) constraint system. Section 4 presents the test data generation by using constraints. Section 5 presents a formalization of our approach in a general setting. Section 6 gives some indications about the implementation of our prototype and gives the results of an experimental evaluation. Lastly we mention some related work before some concluding remarks and perspectives.

2 Background

2.1 A Quick Tour of Focal

Focal is a development environment allowing to develop programs step by step, from the specification phase to the implementation phase. This environment

proposes a language named also Focal and tools to analyze the code -in particular its dependencies-, to compile into various formats -Ocaml executable code, Coq code, html representation and UML class diagrams-, to prove properties (e.g. that the implementation satisfies its specification). In our context a specification is a set of algebraic properties describing relations inputs and outputs of the functions implemented in a Focal program. The Focal language belongs to the family of functional languages but offers structures and mechanisms inspired by object-oriented programming, e.g. inheritance and late binding. The main Focal programming unit, called a species, represents a set of entities, described as a module containing three kinds of items:

- the carrier type, called representation, which is the type of the entities manipulated by the functions of the species; representations can be either abstract (in particular in the early stages of the software life cycle) or concrete;
- the functions which denote the operations allowed on the entities; the functions can be either declarations or definitions: the former ones are just given a name, an arity and a type, the latter ones are also given a body;
- the properties expected for each further implementation; as previously they can be either simple properties or theorems: the first ones are just enounced, a proof is also provided for the second ones.

A species S is said complete when every function in S, inherited or not, is defined and when every property in S, inherited or not, is proved or assumed.

We do not detail further all these features (see [1] for details). We focus in the next section on the subset of the language allowing to define functions and write properties, from which constraints will be extracted.

2.2 Functions and Properties

Functions As said previously, Focal is a wide-spectrum language. Its functional kernel is used to implement the functions of the species. This subset is very close to the functional kernel of ML: recursive (mutual) functions, local binding (**let** $x = e_1$ **in** e_2), conditionals (**if** e **then** e_1 **else** e_2), and pattern-matching expressions (**match** x **with** $pat_1 \rightarrow e_1; \ldots; pat_n \rightarrow e_n$). We assume that programs under test do not contain any higher order or anonymous functions. Indeed testing higher order functions is a difficult problem in particular when generating functions as test data. In the following we consider a Focal program as a set of defined functions.

Moreover, Focal is a strongly typed language. In the following we assume we have access to the type of each variable. We consider values over integer types, cartesian product types and concrete types. Intuitively, a concrete type is defined by a set of typed constructors (with fixed arity). Thus a value of a concrete type is a closed term built from the constructors of the type. For example, the type of binary numbers *binary* is defined by the three constructors { `Null/0`, `One/1`, `Zero/1` } where `One` and `Zero` take an argument of type *binary*. Thus the binary number 10111 is represented by the value `One(Zero(One(One(One(Null)))))`, the binary number 0 is encoded as `Null`. The cartesian product is considered as

a concrete type with a unique constructor which takes as many arguments as types in the product.

Properties are first order formulas. They may contain free variables, the Focal compiler ensures that these variables are well defined somewhere in the species or the inheritance path. See 6.2 for examples of properties. However in this work we focus on the subset of testable properties and even more precisely on elementary properties, the ones which are effectively tested. Below is presented the syntax of the elementary properties:

$$\forall X_1 : \tau_1 \ldots X_n : \tau_n,\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow B_1 \vee \ldots \vee B_m$$

The atomic formulas A_i and B_k are calls to Focal predicates (functions that return a boolean value) with an optional negation, and $\tau_1 \ldots \tau_n$ denote the types of the quantified variables. Calls may be nested here. So, elementary properties are some first order formulas in prenex form without any existential quantifier.

We call the *precondition* (resp. the *conclusion*) of the property $P \equiv \forall X_1 : \tau_1 \ \ldots \ X_n : \tau_n.\ A_1 \Rightarrow \ldots A_n \Rightarrow B_1 \vee \ldots \vee B_m.$, the predicate $Pre(P) = A_1 \wedge \ldots \wedge A_n$ (resp. $Con(P) = B_1 \vee \ldots \vee B_m$).

In our testing approach, properties under test are decomposed, via a rewriting process, into a set of elementary properties. The rewriting system performs very classical logical properties, it is omitted here but is detailed in [6]. In the rest of the paper, we only consider the elementary properties (except in the evaluation section) since they are the ones which are effectively tested, furthermore they are tested separately.

2.3 Testing Framework

So in the following we assume that the FocalTest user has given a species S and the property he is willing to test (PUT), expressed in S or in an ancestor species of S. Testing PUT requires to execute the functions involved (directly or called by an involved function) in the statement. Thus those functions need to be defined in S or inherited. Furthermore the representation must be defined at this stage in order to be able to design test data. No matter whether the proofs are done or not, we do not care about them. Usually the properties the user is interested in are not yet proved, they are artificially and temporarily assumed. For simplicity we impose that S is a complete species. This hypothesis can be relaxed without any major difficulty in order to have a species that only contains the definition of the functions involved in PUT.

A test data is a valuation σ which maps each quantified variable X_i to a value. A positive test data for P is such that $Pre(P)$ is true. The precondition of a property is used to generate the test data and its conclusion is used as an oracle. Indeed to compute the verdict of a test, we evaluate the conclusion with respect to σ. Three results are possible: - *true*, then the verdict is OK, - *false* the verdict is KO, we have found a counter-example that exemplifies the property is

not satisfied for that test data - *an exception is raised*, the tester should decide if the exception is expected or not.

In the following we focus on the test data generation. The construction of the test harness is not detailed, especially in the case of parametrized species because it does not interfere with our purpose (see [6] for the details).

3 Constraint Generation

In this section we present the translation part of the constraint-based test case generation. When translating a precondition, each Focal predicate (and function) involved directly or indirectly (via a call) in the precondition must be translated into an equivalent CLP(FD) program. Thus we present how to translate a function definition and each construction of the language. Then we introduce the translation schema of a precondition into a set of constraints. Semantics of constraints is presented in the next section. However before the translation into constraints, functions are turned into a form that eases the translation. We present this normalization process in the next subsection.

3.1 Normalization of a Function Definition

Each function definition is rewritten in a syntactic form designed to ease the translation into a set of constraints. Each expression and sub-expression will indeed be translated into a constraint. The normalized syntax (named MiniFocal) is presented in Figure 1.

The language MiniFocal expects in many locations variables instead of plain expressions, e.g. arguments in a function call, the condition in a conditional expression or the filtered expression in a pattern-matching expression. Furthermore patterns are linear (a variable occurs only once in the pattern) and they cannot be nested. A basic expression is a variable or a constant: an integer, a boolean or a value built from value constructors.

MiniFocal is in fact an intermediate language in which any real Focal function definition can be translated into. The translation is a simple normalization process where each non variable expression is given a name with a **let** construction. For example, the expression $\mathrm{f}(\mathrm{g}(x+1), 2)$ is translated into the MiniFocal expression **let** $a = x + 1$ **in let** $b = \mathrm{g}(a)$ **in** $\mathrm{f}(b, 2)$.

$$
\begin{array}{lll}
\textit{function} & ::= & \textbf{let}\ [\textbf{rec}]\ \mathrm{f}(x, \ldots, x) = \textit{expr} \\
\textit{expr} & ::= & \textbf{let}\ x = \textit{expr}\ \textbf{in}\ \textit{expr} \mid \textbf{if}\ x\ \textbf{then}\ \textit{expr}\ \textbf{else}\ \textit{expr} \mid \\
 & & \quad \textbf{match}\ x\ \textbf{with}\ \textit{pat} \rightarrow \textit{expr}; \ldots; \textit{pat} \rightarrow \textit{expr} \mid \\
 & & \quad \mathrm{f}(x, \ldots, x) \mid \textit{value} \\
\textit{pat} & ::= & \mathrm{constructor} \mid \mathrm{constructor}(x, \ldots, x) \\
\textit{value} & ::= & n \mid b \mid \mathrm{constructor}(\textit{value}, \ldots, \textit{value}) \mid x
\end{array}
$$

Fig. 1. Syntax of MiniFocal

3.2 Translation of a Function Definition

The function definition (recursive or not) **let** [**rec**] $f(x_1, \ldots, x_n) = e$ is translated in the CLP program $\overline{f}(\overline{r}, \overline{x_1}, \ldots, \overline{x_n}) :- \overline{e}$. Thus a MiniFocal function is translated into a logical predicate with one clause that sets the constraints derived from the body of the function. The argument $\overline{r}$ is the output variable which corresponds to the result of the function. $\overline{e}$ is the constraint resulting from the translation of the body e of the function (according to the translation rules described below). For the translation to be correct, we have to require that all the variables used in a function definition have distinct names. A pre-processing that renames the variables ensures that property. In the following we omit the overlines on objects in the constraint universe when no ambiguity.

The translation of arithmetic and boolean expressions is straightforward. A MiniFocal variable is translated into a CLP(FD) variable. Each variable is associated with a domain constraint depending on its type. We come back on this point in the next section. The constructors of a concrete type are translated into their counterpart in the constraint universe

The translation of the binding expression **let** $x = e_1$ **in** e_2 consists in translating first the defining expression (the output value of this expression is x) then translating the second expression as usual (output value is r). This leads to two constraints, the conjunction of both giving the final constraint. For example, **let** $x = 5 * y$ **in** $x + 3$, is transformed into the constraint $x = 5 * y \wedge r = x + 3$, assuming that the expression is itself bound to the variable r. A function call $f(x_1, \ldots x_n)$ bound to the variable r is translated into $f(r, x_1, \ldots x_n)$.

The conditional expression is treated with a user-defined constraint `ite/3`. The expression **if** x **then** e_1 **else** e_2 is translated into $\texttt{ite}(x, e_1, e_2)$ where e_1 and e_2 are the constraints resulting from the translation of e_1 and e_2 respectively. For example, the expression **if** x **then** $y{+}1$ **else** $f(x)$ is translated into $\texttt{ite}(x, r = y{+}1, f(r, x))$ if r is the output variable for the entire expression. `ite` is a constraint combinator allowing nested constraints.

The pattern-matching expression is translated in the same way, by using another new constraint combinator `match/2`. This predicate takes the name of the matched variable as a first argument and the list of pattern clauses (the pattern and the associated expression) as a second argument. For example, the following expression is translated into the right hand side constraint:

match x **with** Null → 0; One(y) → 1; Zero(z) → 1	`match`(x, [pattern(`Null`, $r = 0$), pattern(`One`(y), $r = 1$), pattern(`Zero`(z), $r = 1$)]).

3.3 Translation of a Precondition

A precondition takes the form $A_1 \wedge \ldots \wedge A_n$. Each A_i is translated into a constraint system, the truth value associated to A_i is named r_i. Let us recall A_i is a function call that may contain nested calls. In this case we first normalize the expression

by naming the non variable sub-expressions with the help of **let** bindings. For example, let A_i be f(x, g(y, z)), it is normalized as **let** a = g(y, z) **in** f(x, a). Then with the previous translation rules we obtain the final constraint $g(a, y, z) \wedge f(r_i, x, a)$ where r_i is the truth value of A_i.

4 Constraint-Based Test Data Generation

Constraint-based test data generation involves solving constraint systems extracted from programs. In this section, we explain the key-point of our approach which consists in exploiting constraint reasoning to implement the constraints we introduced to model faithfully some functional programming features. We first detail the main constraint solving procedures, then, we explain how new dedicated constraints can be defined in FocalTest to deal with conditional and pattern-matching operators. Finally, we present the test data generation procedure and discuss the correction of our constraint model.

4.1 Constraint Solving

A constraint system is composed of variables, built-in constraints and user-defined constraints. A domain is associated to each variable and the constraint solving process aims at pruning the domain. Built-in constraints are directly encoded within the constraint library while user-defined constraints can be added by the user to enrich the capabilities of the constraint solver. Built-in constraints include arithmetical operations such as $+, -, *, min, max \ldots$ while user-defined constraints allow defining new constraints.

Intuitively, a constraint system is solved by the interleaving of three processes, namely *local filtering, constraint propagation* and *labeling. Local filtering* removes values from the variable domains which are not part of the solutions of a given constraint. For example the constraint $x \leq y$ filters all values in the x's domain greater than the maximum bound of the y's domain and all values of the y's domain lower than the minimum bound of the x'domain. For efficiency reasons, this filtering process usually considers only the bounds of large domains and then it is not necessary optimal. This process is local as it considers each constraint apart from the rest of the constraint system. In contrast, *constraint propagation* allows reductions to be propagated throughout the constraint system. Each constraint is examined in turn until a fixpoint is reached. This fixpoint corresponds to a state where no more pruning can be performed. Interestingly, constraint propagation and local filtering are polynomial processes on the number of constraints and variables [8].

The *labeling process* tries to instantiate each variable x to a single value v of its domain by adding a new constraint $x = v$ to the constraint system. Once such a constraint is added, constraint propagation is launched and can refine the domain of other variables. When a variable's domain becomes empty, the constraint system is showed inconsistent (that is the constraint system has no solution), then the labeling process backtracks and other constraints that

bound values to variables are added. To exemplify those processes, consider the following (non-linear) example: x, y in $0..10 \wedge x * y = 6 \wedge x + y = 5$. First the domain of x and y is set to the interval $0..10$, then constraint $x * y = 6$ reduces the domain of x and y to $1..6$ as values $\{0, 7, 8, 9, 10\}$ cannot be part of solutions. Ideally, the process could also remove other values but recall that only the bounds of the domains are checked for consistency and $1 * 6 = 6 * 1 = 6$. This pruning wakes up the constraint $x + y = 5$, that reduces the domain of both variables to $1..4$ because values 5 and 6 cannot validate the constraint. Finally a second wake-up of $x * y = 6$ reduces the domains to $2..3$ which is the fixpoint. The labeling process is triggered and the two solutions $x = 2, y = 3$ and $x = 3, y = 2$ are found. This example works well on integer constraints but an interesting features of those constraint solving processes is that they can be used to check the consistency of other kind of constraints. In particular, constraints over the Herbrand domain (a.k.a. Prolog terms and unification) can be solved using similar techniques.

4.2 User-Defined Constraints

In most constraint solver implementation, the user can define new constraints with the help of dedicated interfaces. Defining new constraints requires to instantiate the following three points:

- A constraint interface including a name and a set of variables on which the constraint holds. This is the entry point of the newly introduced constraint;
- The wake-up conditions. A constraint can be awaken when either the domain of one of its variables has been pruned, or one of its variables has been instantiated, or a new constraint related to its variables has been added. According to the system used, different wake-up conditions can be exploited.
- An algorithm to call on wake-up. The purpose of this algorithm is to check whether the constraint is consistent[4] or not with the new domains of variables and also to prune the domains.

4.3 MiniFocal Constraints

MiniFocal constraints, i.e. constraints obtained by translation of MiniFocal functions and properties, include simple equality and disequality constraints over variables of concrete types, numerical constraints over finite domain variables and also user-defined constraints used to model the control– and data– flow operations of the language.

Finite domain variables are generated from MiniFocal variables using their type as a specification of their domain. For example, a MiniFocal integer variable will be translated with an FD variable with domain $0..2^{32} - 1$. Variables of concrete type have a specific domain that depends on the MiniFocal program. For example, if variable x has type *binary* then $\mathrm{dom}(x) = \{\texttt{null}, \texttt{one}, \texttt{zero}\}$ and a disequality constraint such as $x \neq \texttt{zero}$ will prune $\mathrm{dom}(x)$ to be $\{\texttt{null}, \texttt{one}\}$.

[4] if there is a solution of the constraint system

When the head of a term is instantiated with a constructor, the domain of each of its arguments is modified accordingly. For example, if x whose type is *list of integers* is instantiated with the constructor `cons` (which takes two arguments, an integer and a list of integers), then the constraint $x = \texttt{cons}(y, z)$ is introduced with $\text{dom}(y) = min(\text{int})..max(\text{int})$ and $\text{dom}(z) = \{\texttt{nil}, \texttt{cons}\}$.

This section also details the semantics of the two user-defined constraints `ite/3` and `match/2` introduced in Sec. 3. Both constraints are based on a specific test called an *entailment test*. This test exploits constraint propagation and refutation to detect the entailment of a given constraint by a constraint-store. Let S be a constraint system and c be a constraint, the entailment test computes the negation of c (noted $\neg c$), adds it to S and tries to show inconsistency by launching constraint propagation and local filtering. Note that the labeling process is not launched at this step as it will be too costly to enumerate all the domains just to check entailment. If the constraint solver finds that $S \wedge \neg c$ has no solution then c is shown to be entailed by S. Otherwise, nothing can be deduced as constraint propagation and local filtering only do not ensure full consistency.

`ite` Operator As said above, each MiniFocal conditional expression is translated with the `ite` constraint, hence such constraints can be nested. The semantics of the constraint $\texttt{ite}(c, C_1 \wedge \ldots \wedge C_n, C'_1 \wedge \ldots \wedge C'_m)$ is defined by four rules. Here are the first two:

- if c is entailed by the constraint system, then the constraint `ite` rewrites to $C_1 \wedge \ldots \wedge C_n$;
- if $\neg c$ is entailed by the constraint system then the constraint `ite` rewrites to $C'_1 \wedge \ldots \wedge C'_m$.

Both rules model the operational interpretation of any MiniFocal conditional expression. These rules allows for a forward chaining reasoning.
And the next two rules of `ite` are:

- If $\neg(c \wedge C_1 \wedge \ldots \wedge C_n)$ is entailed by the constraint system then the constraint `ite` rewrites to $\neg c \wedge C'_1 \wedge \ldots \wedge C'_m$;
- If $\neg(\neg c \wedge C'_1 \wedge \ldots \wedge C'_m)$ is entailed by the constraint system the constraint `ite` rewrites to $c \wedge C_1 \wedge \ldots \wedge C_n$

These two rules allow for a backward reasoning. They can be interpreted as follows: if one of the branches of the conditional expression is shown to be incompatible with the rest of the constraint system then the `ite` constraint is replaced by the constraint associated to the other branch.

Consider the following constraint system $Y = 0, \texttt{ite}(X = 3, Z = 0, Z = 1), \texttt{ite}(Z > 0, Y = X, Y = 5)$. with $X \in 0..2^{32} - 1$ and $Z \in 0..2^{32} - 1$. From the first `ite` constraint, nothing can be deduced as none of the four rules can be applied and the constraint simply suspends. On the contrary, the fourth rule of the second `ite` constraint applies as $Y = 5$ is incompatible with $Y = 0$ and the store rewrites to $Y = 0, \texttt{ite}(X = 3, Z = 0, Z = 1), Z > 0, Y = X$. Hence, we

get that $X = 0$, and $Z \in 1..2^{32} - 1$. Pruning the domain Z awakes the first `ite` and its third rule applies as $Z = 0$ is incompatible with $Z \in 1..2^{32} - 1$. Finally, the constraint system rewrites to $Y = 0, X \neq 3, Z = 1, Y = X$ and the single solution $(X = 0, Y = 0, Z = 1)$ is found.

`match` Operator Similarly to the `ite` operator, the `match` operator is implemented as a global constraint. Let x be a variable, $pat_1, \ldots, pat_n$ be n patterns and $C_1, \ldots, C_n$ be n constraints, then the constraint `match`$(x, [(pat_1, C_1), \ldots, (pat_n, C_n)])$ has the following semantics:

- if $n = 0$ then the `match` constraint `fails` (no pattern) ;
- if $n = 1$ then the `match` constraint rewrites to $x = pat_1 \wedge C_1$;.
- if $\exists i$ in $1..n$ such that $x = pat_i$ is entailed by the constraint system, then the `match` constraint rewrites to C_i ;
- if $\exists i$ in $1..n$ such that $\neg(x = pat_i \wedge expr_i)$ is entailed by the constraint system then the `match` constraint rewrites to `match`$(x, [(pat_1, C_1), \ldots, (pat_{i-1}, C_{i-1}), (pat_{i+1}, C_{i+1}), (pat_n, C_n)])$.

The two former rules implement trivial terminal cases. The third rule implements forward deduction w.r.t. the constraint system while the fourth rule implements backward reasoning. Note that these two latter rules use (don't-care) nondeterministic choices to select the pattern to explore first.

Example 1. Consider the following example of `match` constraint:
`match`$(l,$ [pattern(`nil`, $r = 0$), pattern(`cons`(x, y), $r = x + 10)$]) where $dom(l) = \{$`nil`, `cons`$\}$ and $dom(r) = 6..14$. As constraint $\neg(l =$ `nil`$\wedge r = 0)$ is entailed by the current domains when the fourth rule is examined ($r = 0$ and $dom(r) = 6..14$ are incompatible), the constraint rewrites to `match`$(l,$ [pattern(`cons`(x, y) $r = x + 10)$]). Then, the second rule applies as it remains only a single pattern: $l =$ `cons`$(x, y) \wedge r = x+10$. Finally, pruning the domains of variables to $r \in 6..14$, $x \in -4..4$, $y \in \{$`nil`, `cons`$\}$ and $l =$ `cons`(x, y) ends up the solving process.

4.4 Test Data Labeling

The final part of the constraint solving involves variable labeling. In our framework, we have to instantiate variables of two kinds: integer variables and variables of concrete MiniFocal types. As these latter variables are structured and involve other variables (such as in the above example of list of integers), we decide to instantiate them first. Note that labeling a variable can awake other constraints that hold on this variable and if a contradiction is found, then the labeling process backtracks to another value or variable. Labeling integer variables requires to define which variable and value to enumerate first. Several heuristics exist such as labeling first the variable with the smallest domain (*first-fail* principle) or the variable on which the most constraints hold. It is worth noticing that once all the variables have been instantiated and the constraints have been verified then we hold a test datum that satisfies the testing objective.

4.5 Correctness, completeness and termination

Total correction of our constraint model implies showing correctness, completeness and termination. If we make the strong hypothesis that CLP(FD) predicates correctly implement arithmetical MiniFocal operators and that the underlying constraint solver is correct, then the correctness of our model is guaranteed, as the deduction rules of `ite/3` and `match/2` directly flow from the operational semantics of conditional and pattern matching in Focal. Completeness comes from the completeness of the labeling process in CLP(FD). In fact, as soon as every possible test data is possibly enumerated during the labeling step, any solution will be eventually found. But completeness comes at the price of efficiency and preserving it is not indispensable in our context as we are only interested in finding test data satisfying PUT. A proof of the correctness and the completeness has been written (independently from Prolog). It required to specify the formal semantics of the Focal functional core, the semantics of constraints, to define formally the translation and the notion of solution of a constraint system derived from a Focal expression. It is detailed in the next section.

Our approach has no termination guarantee, hence it is only a semi-correct procedure. To leverage the problems of non-termination, we introduced several mechanisms such as time-out, memory-out and various bounds on the solving process. When such a bound is reached, other labeling heuristics are tried in order to avoid the problem. Note however that enforcing termination yields loosing completness as this relates to the Halting problem.

5 Formalisation

5.1 MiniFocal Language

MiniFocal Semantics We first introduced some definitions. More precisely it consists mainly in the definition of the environments used in the semantics.

Definition 1 (Function symbols). *We denote by F the set of MiniFocal function symbols. An element of this set is denoted f.*

Definition 2 (Function environment). *The syntax of a function environment is:*

$$\begin{aligned}\mathcal{E}_{\mathsf{f}} ::= & \emptyset \\ & \mid \mathcal{E}_{\mathsf{f}'}(\mathsf{f} \leftarrow < x_1, \ldots, x_n \rightsquigarrow e >)\end{aligned}$$

An environment binds a function symbol to a MiniFocal closure.

We remark a closure does not embed an environment as usual. In a MiniFocal program, all functions are recursive and so share the same environment.

Definition 3 (Variable environments). *A variable environment is defined in the same way. It binds each variable to a value.*

We can now define the semantics of the MiniFocal expressions with the help of inference rules:

$$\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e \rhd v$$

It means, the expression e evaluates to v wrt the environment $\mathcal{E}_x$ and the function environment $\mathcal{E}_\mathsf{f}$.

Figure 2 shows the different semantic rules of MiniFocal expressions. These rules are the usual ones, very close to the ML core language, we don't detail them.

The following theorems express some useful semantic properties. The two first ones are classical. they are the weakening theorem and reinforcement theorem.

Theorem 1. *Let $\mathcal{E}_x$ a variable environment and x' a variable non free in an expression e. Let $\mathcal{E}_\mathsf{f}$ a function environment and two values v and v'.*

$$\textit{If } (\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e \rhd v) \textit{ then } (\mathcal{E}_x \oplus (x' \leftarrow v'); \mathcal{E}_\mathsf{f} \vdash_N e \rhd v)$$

Proof. By induction on the derivation of $\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e \rhd v$.

Theorem 2. *Let $\mathcal{E}_x$ a variable environment, x' a variable non free in an expressions e, a function environment $\mathcal{E}_\mathsf{f}$ and two values v and v'.*

$$\textit{If } (\mathcal{E}_x \oplus (x' \leftarrow v'); \mathcal{E}_\mathsf{f} \vdash_N e \rhd v) \textit{ then } (\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e \rhd v)$$

Proof. By induction on e.

The two following theorems are similar but hold on function environments.

Theorem 3. *Let $\mathcal{E}_\mathsf{f}$ and $\mathcal{E}'_\mathsf{f}$ two function environments such as $\mathcal{E}_\mathsf{f} \subseteq \mathcal{E}'_\mathsf{f}$, a variable environment $\mathcal{E}_x$, an expression e and a value v.*

$$\textit{If } (\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e \rhd v) \textit{ then } (\mathcal{E}_x; \mathcal{E}'_\mathsf{f} \vdash_N e \rhd v)$$

Proof. By induction on the derivation of $\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e \rhd v$.

Theorem 4. *Let $\mathcal{E}_x$ a variable environment, $\mathcal{E}_\mathsf{f}$ a function environment, an expression e, a variable x and two value v and v'.*

$$\textit{If } (\mathcal{E}_x, (x \leftarrow v); \mathcal{E}_\mathsf{f} \vdash_N e \rhd v') \textit{ then } (\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e[x \leftarrow v] \rhd v')$$

Proof. By induction on the derivation.

The last theorem determines the minimum set of variables that should be defined by an environment for evaluating an expression.

Theorem 5. *Let a variable environment $\mathcal{E}_x$, a function environment $\mathcal{E}_\mathsf{f}$, an expression e and a value v.*

$$\textit{if } \mathcal{E}_x, \mathcal{E}_\mathsf{f} \vdash_N e \rhd v \textit{ then } \mathcal{E}_x \textit{ binds all free variable of } e.$$

Proof. By induction on e.

$$\frac{\mathcal{E}_x(x) = v}{\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N x \rhd v}\ \textsc{NVar}$$

$$\frac{\mathcal{E}_x(x_1) = v_1 \qquad \ldots \qquad \mathcal{E}_x(x_n) = v_n}{\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N C(x_1, \ldots, x_n) \rhd C(v_1, \ldots, v_n)}\ \textsc{NConstructeur}$$

$$\frac{}{\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N i \rhd i}\ \textsc{NEntier}$$

$$\frac{\begin{array}{c}\mathcal{E}_x(x) = \textbf{true} \\ \mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e_1 \rhd v_1\end{array}}{\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2 \rhd v_1}\ \textsc{NIf\_true}$$

$$\frac{}{\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N C^0 \rhd C^0}\ \textsc{NConstante}$$

$$\frac{\begin{array}{c}\mathcal{E}_x(x) = \textbf{false} \\ \mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e_2 \rhd v_2\end{array}}{\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N \textbf{if } x \textbf{ then } e_1 \textbf{ else } e_2 \rhd v_2}\ \textsc{NIf\_false}$$

$$\frac{\begin{array}{c}\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N e_1 \rhd v_1 \\ \mathcal{E}_x \oplus (x \leftarrow v_1); \mathcal{E}_\mathsf{f} \vdash_N e_2 \rhd v_2\end{array}}{\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N \textbf{let } x = e_1 \textbf{ in } e_2 \rhd v_2}\ \textsc{NLet}$$

$$\frac{\begin{array}{c}\mathcal{E}_x(x_1) = v_1 \qquad \ldots \qquad \mathcal{E}_x(x_n) = v_n \\ \mathcal{E}_\mathsf{f}(\mathsf{f}) = <x_1, \ldots, x_n \rightsquigarrow e_f> \\ (x_1, v_1), \ldots, (x_n, v_n); \mathcal{E}_\mathsf{f} \vdash_N e_f \rhd v_f\end{array}}{\mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N \mathsf{f}(x_1, \ldots, x_n) \rhd v_f}\ \textsc{NApply}$$

$$\frac{\begin{array}{c}\mathcal{E}_x(x) = v \qquad \mathcal{F}(pat_i, v) = \mathcal{E}_x{}' \\ \forall j < i,\ \mathcal{F}(pat_j, v) = \texttt{fail} \qquad \mathcal{E}_x, \mathcal{E}_x{}'; \mathcal{E}_\mathsf{f} \vdash_N e_i \rhd v_i\end{array}}{\begin{array}{l}\textbf{match } x \textbf{ with} \\ \mathcal{E}_x; \mathcal{E}_\mathsf{f} \vdash_N \quad | \ pat_1 \rightarrow e_1 \\ \qquad\qquad \vdots \qquad\qquad \rhd v_i \\ \quad | \ pat_n \rightarrow e_n\end{array}}\ \textsc{NFiltrage}$$

Fig. 2. Semantic of MiniFocal expressions

5.2 Constraint language

We present now the target constraint language. This language is powerful enough for expressing the precondition of our properties. We present also a predicate that decidew whether an affectation is a solution of a constraint system or not.

$$
\begin{array}{lll}
\sigma ::= c & \text{unique constraint} & \\
\quad \mid c, \sigma & \text{constraint conjunction} & \\
\quad \mid \sigma_1 \wedge \sigma_2 & \text{store conjunction} & \\
 & & \\
c ::= X =_{fd} a & & \text{integer equality} \\
\quad \mid X \neq_{fd} a & & \text{integer inequality} \\
\quad \mid X =_h t & & \text{concrete type equality} \\
\quad \mid X \neq_h t & & \text{concrete type inequality} \\
\quad \mid \mathtt{f}(X_1, \ldots, X_n) & & \text{function call} \\
\quad \mid \begin{array}{l} \mathtt{match}(X, [\ \mathtt{pattern}(pat, \sigma), \\ \qquad \ldots, \\ \qquad \mathtt{pattern}(pat, \sigma)], \sigma) \end{array} & & \text{pattern matching constraint} \\
\quad \mid \mathtt{ite}(X, \sigma, \sigma) & & \text{if constraint} \\
 & & \\
pat ::= \mathtt{c}(X_1, \ldots, X_n) & \text{pattern} & \\
 & & \\
a ::= i & \text{direct value} & \\
\quad \mid X & \text{integer value} & \\
\quad \mid op(X_1, \ldots, X_n) & \text{arithmetic operator} & \\
 & & \\
t ::= X & \text{concrete variable} & \\
\quad \mid \mathtt{c}(X_1, \ldots, X_n) & \text{head matching} &
\end{array}
$$

Fig. 3. Syntax of the constraint language

Syntax The syntax of the constraint language is given in Figure 3. We denote a constraint system by σ, it is called a store of constraints. A store is a conjunction of elementary constraints. The elementary constraints are:

- the integer equality, $X =_{fd} a$. It imposes the values in both sides of equality to be identical. On the rhs, we accept: a integer value i; a constraint variable X; an operator applied to a set of variables $op(X_1, \ldots, X_n)$
- the integer inequality, it is similar to the integer equality but imposes both sides to be different values.
- the concrete type equality $X =_h t$. It acts as the integer equality except it occurs over concrete values.
- the concrete type inequality.

– the constraint $\mathtt{f}(X_1, \ldots, X_n)$ corresponds to a call of a constraint function (we define constraint function below). It is a special constraint which is, during the resolution process, replaced by a set of constraints over the variable $X_1, \ldots, X_n$.
– the constraints $\mathtt{match}(X, \ldots)$ and $\mathtt{ite}(X, \sigma, \sigma)$ are the operators defined in 4.3.

Example 2. For example, the system below imposes the variable R to be of the form $\mathtt{c}(2 * X, \mathtt{cons}(X, nil)$ where X is an integer value.

$$R = \mathtt{c}(X_1, R_1), R_1 = \mathtt{cons}(X_2, R_2), R_2 = \mathtt{nil}, X_1 = 2 * X_2$$

Definition 4 (Conjunction of stores). *Let two constraint stores σ_1 and σ_2. The conjunction of σ_1 and σ_2 is denoted $\sigma_1 \wedge \sigma_2$. $\wedge$ is a combinator, here is his definition:*

$$c \wedge \sigma = c, \sigma$$
$$(c, \sigma_1) \wedge \sigma_2 = c, (\sigma_1 \wedge \sigma_2)$$

Variable Domain Each variable in the constraint system has a domain. We define here the domain for the integer and concrete type variable. The domain is the set of values that can instantiate the variable. For example, if X is an integer list, his domain is:

$$\{\mathtt{nil}, \mathtt{cons}(I_1, \mathtt{nil}), \mathtt{cons}(I_2, \mathtt{cons}(I_3, \mathtt{nil})), \ldots\}$$

Where the I_1 are integers.

$\theta ::= \emptyset$ empty environement
$\quad | \ \theta, (\tau, [\delta_t, \ldots, \delta_t])$ liste of types

$\delta_t ::= \delta_c$ constructor
$\quad | \ \delta_c, \delta_t$ constructors

$\delta_c ::= \mathtt{c}$ constant
$\quad | \ \mathtt{c}(carg_1, \ldots, carg_n)$ parametrized parameter

$carg ::= \alpha$ type argument
$\quad | \ \tau$ type

$\tau ::= \iota$ type
$\quad | \ \iota(\alpha_1, \ldots, \alpha_n)$ parametrized type

Fig. 4. Syntax of the type definition environments

Constraints and Types Figure 4 shows the syntax of the type definition environments of our language. Such an environment associates a type symbol to the list of his constructors. A constructor is defined by a name and the type of his expected arguments.

Example 3. Let θ be:

$$(\texttt{list}(\alpha), [\texttt{nil}, \texttt{cons}(\alpha, \texttt{list}(\alpha))]), (\texttt{option}(\alpha), [\texttt{none}, \texttt{some}(\alpha)])$$

θ defines two types, the first one is the definition of lists. The elements of the lists are of type α. The second defined type is the option type.

Definition 5 (Well Formed Environnement). *A type definition environment is well formed if for each definition the set of free types is empty.*

In the following, we consider only well formed environments.

Definition 6 (Constructor set). *Let a type τ and a type definition environment θ, the constructor set of τ is denoted by $\mathcal{C}(\tau, \theta)$.*

When no ambiguity, we will omit the mention of the type definition environment and write $\mathcal{C}(\tau)$.

Example 4. Let θ the type definition environment of 3, the constructor set of the list is:

$$\mathcal{C}(\texttt{list}(\texttt{int})) = \{\texttt{nil}, \texttt{cons}\}$$

Definition 7 (Constructor instantiation). *Let a type τ and a constructor $\texttt{c} \in \mathcal{C}(\tau)$. The function $\mathcal{C}_{args}(\tau, \texttt{c})$ returns list of arguments of $\texttt{c}$ where the types variables are replaced by their definition.*

Example 5. We take the type definition environment of 3. We have:

$$\mathcal{C}_{args}(\texttt{list}(\texttt{int}), \texttt{cons}) = [\texttt{int}, \texttt{list}(\texttt{int})]$$
$$\mathcal{C}_{args}(\texttt{option}(\texttt{list}(\texttt{int})), \texttt{some}) = [\texttt{list}(\texttt{int})]$$

Type Environment

Definition 8 (Type Environnement). *A types environment associates each variable to his type:*

$$\begin{aligned} \Gamma ::= & \emptyset \\ & \mid \Gamma, (X : \tau) \end{aligned}$$

We will consider below that all variables are typed in an environment Γ.

Integer Variables

Definition 9 (Integer variable domain). *Let X an integer variable. The domain of X is defined by the function DOM_i. We have $DOM_i(X) \subseteq \mathcal{N}$ where $\mathcal{N}$ is the set of possible integer values.*

Integer constraint As said in the presentation of the syntax, we have two constraints over integers, equality and inequality.

Concrete Variables

Definition 10 (Concrete domain). *Let a variable X and a type τ such as $\Gamma(X) = \tau$. The domain of X is given by the function DOM_h. DOM_h returns a set of constructor symbols. So we have $DOM_h(X) \subseteq \mathcal{C}(\tau)$.*

Intuitively, the domain of a concrete type is a set of constructor names. For a variable X and a constructor $\mathtt{c}$ such as $\mathtt{c} \in DOM_h(X)$, a possible value for X is a term $\mathtt{c}(\ldots)$ where the arguments are terms of the expected type.

Example 6. If B is a boolean variable, his domain is:

$$DOM_h(B) = \mathtt{true}, \mathtt{false}$$

When the domain is a singleton, for example, $DOM_h(X) = \{\mathtt{c}\}$, the variable is implicitly instantiated to a term with head $\mathtt{c}$. So, the value of X becomes $\mathtt{c}(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are fresh variables such as $\mathcal{C}_{args}(\tau, \mathtt{c}) = [\tau_1, \ldots, \tau_n]$ and $\forall i \in [\![1, n]\!], DOM_h(X_i) = \mathcal{C}(\tau_i)$ (for the concrete variables) and $DOM_i(x_i) = \mathcal{N}$ (for the integer variables). The type environment is augmented with the variables and their respected types.

Constraint over Concrete Types We dispose of two constraints over the concrete types.

Definition 11 (Equality). *Let X, Y two constraint variables. The equality operator is denoted by $=_h$. The constraint $X =_h Y$ imposes the variables X and Y have the same value.*

Definition 12 (Inequality). *Let X, Y two constraint variables. The inequality operator is written $\neq_h$. The constraint $X \neq_h Y$ imposes the variables X and Y have different values.*

Clauses

Definition 13 (Clause Environnements). *The clause environments are defined by the following syntax:*

$$\begin{aligned} \mathcal{E}_{\mathtt{cl}} ::= & \emptyset \\ & \mid \mathcal{E}_{\mathtt{cl}}, (\mathtt{f}, <X_1, \ldots, X_n \rightsquigarrow \sigma > \Gamma) \end{aligned}$$

A clause environment associates a function symbol to a type environment and an abstraction.

Informally, the clauses are the constraints counterpart to the functions in programming languages. It defines a set of constraints over variables. The clause definition $(\mathtt{f}, <X_1, \ldots, X_n \rightsquigarrow \sigma>)$ means $\mathtt{f}$ leads to the constraint σ over the variables $X_1, \ldots, X_n$. The free variables of σ are existentially quantified. It means the constraint $\mathtt{f}(X'_1, \ldots, X'_n)$ is verified for a variable affectation of $X'_1, \ldots, X'_n$ if and only if there exist values for the free variables of σ such as σ is verified.

Example 7. A constraint which imposes a list to contain a unique integer value can be defined as follow:

$$\begin{array}{l} \mathtt{singleton}, <I, L \rightsquigarrow \sigma> \\ \qquad\qquad \text{avec} \\ \sigma = \{Y =_h \mathtt{nil}, L =_h \mathtt{cons}(I, Y)\} \end{array}$$

Affectation

Definition 14 (Affectation). *An affectation is a function/environment which associates variables to a value of their domain. The syntax is:*

$$\begin{array}{rl} \mathcal{A} ::= & \emptyset \\ & \mid \mathcal{A}, (X, v) \end{array}$$

The values are the concrete values and the integer values:

$$\begin{array}{rll} v ::= & i & \textit{valeur entire} \\ & \mid \mathtt{c}(v_1, \ldots, v_n) & \textit{valeur construit} \\ & \mid \mathtt{a} & \textit{valeur constante} \end{array}$$

Definition 15 (Partial/Total Affectation). *An affectation over a store σ is* partial *if it defines the value of not all free variables of σ. An affectation is* total *if it defines values for all the variables of σ.*

Intuitively, an affectation permits to denote a solution of a constraint system. Let us give some other definitions.

Definition 16 (Violation). *Let an elementary constraint c. An affectation $\mathcal{A}$* violates *the constraint c if all the variables of c are instantiated by $\mathcal{A}$ and if c is not verified, according to the valuation of its variables.*

We can now extend the notion of violation of constraint over a store of constraints.

Definition 17 (Affectation Consistency). *Let a store σ. An affectation $\mathcal{A}$ is consistent w.r.t. σ if it doesn't violate constraints of σ. If $\mathcal{A}$ violates at least one constraint of σ, $\mathcal{A}$ is inconsistent.*

Definition 18 (Solution). *Let a store σ. An affectation $\mathcal{A}$ is a solution of σ if it is total and consistent w.r.t. σ.*

Solution Checking of a Constraint System The checking of a solution is defined by a predicate which takes two affectations $\mathcal{A}$ and $\mathcal{A}'$and a constraint system as arguments. It is required as a precondition the constraint systems is obtained by translation of the MiniFocal programs. We don't handle inequality constraints ($\neq_{fd}$ and $\neq_h$) because the translation function doesn't generate such constraints.

When the predicate is verified, the affectation $\mathcal{A}'$ is total and consistent w.r.t. the input constraint system. It overloads the input affectation $\mathcal{A}$ with new valuations when the input affectation is not total.

The returned affectation $\mathcal{A}'$ is justified for two reasons.

Firstly, the free variables of a clause are existentially quantified. Assume we have a constraint $f(X, Y)$ and an affectation $\mathcal{A} = (X, 4), (Y, 7)$. If we want to check whether $\mathcal{A}$ is a solution of $f(X, Y)$, we unfold the definition of f and introduce variables that are not defined by $\mathcal{A}$ (the free variables of f definition). We have to define the predicate for checking in this case if $\mathcal{A}$ is effectively a solution. Of course, we can't be complete and decide all constraint systems. But the predicate we define is sufficient for deciding the solution of the constraint systems obtained from a MiniFocal program.

Secondly, we need to define the equivalence between an affectation and a variable environment for the correctness and completeness of the translation. Let us consider an expression e and the corresponding constraint σ. The set of free variables of both are not identical. The variables bound by the **let** constructions are not bound in σ. In the correctness/completeness theorems, we express a solution as a partial affectation that defines values over free variables in an expression e (and we consider, of course, the affectation over constraint system obtained from e). Thus, the predicate takes as input a partial affectation and we set in an affectation over the free variables of the considered expression.
the definition of the predicate uses inference rules of the form

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'$$

It means the affectation $\mathcal{A}$ is consistent w.r.t. the store σ and returns the new affectation $\mathcal{A}'$. $\mathcal{A}$ The definition of the predicate is defined in Figures 5 and 6.

As said previously, the predicate doesn't decide the consistency of all constraint systems. For example, let the constraint system $X =_{fd} Y$ and the (consistent) affectation $\mathcal{A} = (X, 3)$, the predicate fails at checking the consistency because the only applicable rule is EqFdX and it imposes Y to be defined in $\mathcal{A}$.

Theorem 6. *Let two affectations $\mathcal{A}$, $\mathcal{A}'$ and a constraint system σ. If we have $\mathcal{A} \vdash_S \sigma \longmapsto \mathcal{A}'$ then $\mathcal{A}'$ is total and consistent w.r.t. σ.*

Proof. by induction on the derivation $\mathcal{A} \vdash_S \sigma \longmapsto \mathcal{A}'$. We remark at each stage, $\mathcal{A}'$ is defined from $\mathcal{A}$ with valuations for the variables not defined in $\mathcal{A}$.

Theorem 7. *Let two affectations $\mathcal{A}_1$, $\mathcal{A}_2$ and a system constraint such as*

$$\mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$$

$$\frac{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S c \longmapsto \mathcal{A}_1 \qquad \mathcal{A}_1;\mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S c,\sigma \longmapsto \mathcal{A}_2}\ \textsc{Conjonct1}$$

$$\frac{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \qquad \mathcal{A}_1;\mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}_2}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \wedge \sigma_2 \longmapsto \mathcal{A}_2}\ \textsc{Conjonct2}$$

$$\frac{\mathcal{A}(X) = i}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} i \longmapsto \mathcal{A}}\ \textsc{EqFdI}$$

$$\frac{X \notin \mathcal{A}}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} i \longmapsto \mathcal{A} \oplus (X \leftarrow i)}\ \textsc{EqFdI}$$

$$\frac{\mathcal{A}(X) = \mathcal{A}(Y)}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} Y \longmapsto \mathcal{A}}\ \textsc{EqFdX}$$

$$\frac{X \notin \mathcal{A} \qquad Y \in \mathcal{A}}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} Y \longmapsto \mathcal{A} \oplus (X \leftarrow \mathcal{A}(Y))}\ \textsc{EqFdX}$$

$$\frac{\mathcal{A}(X) = [\![op(\mathcal{A}(X_1),\ldots,\mathcal{A}(X_n)]\!]}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} op(X_1,\ldots,X_n) \longmapsto \mathcal{A}}\ \textsc{EqFdOp}$$

$$\frac{X \notin \mathcal{A} \qquad v = [\![op(\mathcal{A}(X_1),\ldots,\mathcal{A}(X_n)]\!]}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} op(X_1,\ldots,X_n) \longmapsto \mathcal{A} \oplus (X,v)}\ \textsc{EqFdOp}$$

$$\frac{\mathcal{A}(X) = \mathtt{c}(\mathcal{A}(X_1),\ldots,\mathcal{A}(X_n))}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_h \mathtt{c}(X_1,\ldots,X_n) \longmapsto \mathcal{A}}\ \textsc{EqHC}$$

$$\frac{X \notin \mathcal{A} \qquad v = \mathtt{c}(\mathcal{A}(X_1),\ldots,\mathcal{A}(X_n))}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} op(X_1,\ldots,X_n) \longmapsto \mathcal{A} \oplus (X,v)}\ \textsc{EqHC}$$

$$\frac{\mathcal{A}(X) = \mathcal{A}(Y)}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_h Y \longmapsto \mathcal{A}}\ \textsc{EqHX}$$

$$\frac{X \notin \mathcal{A} \qquad Y = [\![op(\mathcal{A}(X_1),\ldots,\mathcal{A}(X_n))]\!]}{\mathcal{A};\mathcal{E}_{\mathtt{cl}} \vdash_S X =_{fd} op(X_1,\ldots,X_n) \longmapsto \mathcal{A} \oplus (X,Y)}\ \textsc{EqHX}$$

Fig. 5. Predicate that checks if an affectation is a solution of a constraint system

$$\frac{\begin{array}{c}\mathcal{E}_{\mathtt{cl}}(\mathtt{f}) = <X'_1, \ldots, X'_n \rightsquigarrow \sigma> \\ (X'_1 \leftarrow \mathcal{A}(X_1)), \ldots, (X'_n \leftarrow \mathcal{A}(X_n)); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'\end{array}}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathtt{f}(X_1, \ldots, X_n) \longmapsto \mathcal{A}} \textsc{Call}$$

$$\frac{\mathcal{A}(X) = \mathtt{true} \qquad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathtt{ite}(X, \sigma_1, \sigma_2) \longmapsto \mathcal{A}'} \textsc{Ite\_true}$$

$$\frac{\mathcal{A}(X) = \mathtt{false} \qquad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}'}{\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathtt{ite}(X, \sigma_1, \sigma_2) \longmapsto \mathcal{A}'} \textsc{Ite\_false}$$

$$\frac{\begin{array}{c}\mathcal{A}(X) = \mathtt{c}_k(\mathcal{A}(X_1^k), \ldots, \mathcal{A}(X_{n_k}^k)) \qquad 1 \leq k \leq i \\ \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_k \longmapsto \mathcal{A}'\end{array}}{\begin{array}{l}\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \\ \quad \mathtt{match}(X, [\\ \qquad \mathtt{pattern}(X =_h \mathtt{c}_1(X_1^1, \ldots, X_{n_1}^1), \sigma_1), \\ \qquad \vdots \\ \qquad \mathtt{pattern}(X =_h \mathtt{c}_i(X_1^i, \ldots, X_{n_i}^i), \sigma_i) \ ], \\ \qquad \sigma_{i+1})\end{array} \longmapsto \mathcal{A}'} \textsc{Match\_pat}$$

$$\frac{\begin{array}{c}\mathcal{A}(X) \neq \mathtt{c}_k(\mathcal{A}(X_1^k), \ldots, \mathcal{A}(X_{n_k}^k)) \qquad 1 \leq k \leq i \\ \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_{i+1} \longmapsto \mathcal{A}'\end{array}}{\begin{array}{l}\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \\ \quad \mathtt{match}(X, [\\ \qquad \mathtt{pattern}(X =_h \mathtt{c}_1(X_1^1, \ldots, X_{n_1}^1), \sigma_1), \\ \qquad \vdots \\ \qquad \mathtt{pattern}(X =_h \mathtt{c}_i(X_1^i, \ldots, X_{n_i}^i), \sigma_i) \ ], \\ \qquad \sigma_{i+1})\end{array} \longmapsto \mathcal{A}'} \textsc{Match\_other}$$

Fig. 6. Predicate that checks if an affectation is a solution of a constraint system

. Let a variable X and a value $\mathtt{v}$ *such as* $\mathcal{A}_2(X) = \mathtt{v}$ *then* $\mathcal{A}_1 \oplus (X \leftarrow \mathtt{v}); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$.

Proof. By structural induction on the constraint system. σ.

Theorem 8. *Let two affectations* $\mathcal{A}_1$, $\mathcal{A}_2$ *and a constraint system such as*

$$\mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$$

. Let an expression e *and a variable* X, *if* $\mathcal{T}_x; X \vdash_C e \longmapsto \sigma$, *then* $\{X\} \lhd \mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$.

Proof. By structural induction on the constraint system σ.

5.3 MiniFocal Program Translation

Variables, Patterns and Function Symbols We first introduce the translation of the MiniFocal variables.

Definition 19 (Fresh variables). *We suppose the existence of a set of fresh variable symbols denoted by* $\mathcal{F}reshC$.

Definition 20 (Variables Translation). *The variable translation environment associates a MiniFocal variable to a constraint variable. We denote a variable translation environment by* $\mathcal{T}_x$. *Let a constraint variable* X, *if* $\mathcal{T}_x$ *associates the MiniFocal variable* x *to* X *we denote:* $X = \mathcal{T}_x(x)$.

This function $\mathcal{T}_x$ expresses the link between a MiniFocal program and its translation. It's used during the translation. Intuitively, when a variable x is introduced by a **let**, x is associated to a fresh variable X. Each further occurrence of x is translated to this X.

Definition 21 (Type constructor). *We translate MiniFocal constructors into term symbols with an injective function. The translation is syntactic. A MiniFocal constructor* C *is translated into the term symbol* $\mathtt{c}^n$.

Definition 22 (Pattern translation). *We define* $\mathcal{T}_p$ *as the function that translates a pattern into a term in the constraint universe:*

$$\mathcal{T}_p(C^0) = \mathtt{c}^0 \qquad \mathcal{T}_p(C^n(x_1, \ldots, x_n) = \mathtt{c}^n(\mathcal{T}_x(x_1), \ldots, \mathcal{T}_x(x_n))$$

Definition 23 (Function Symbol). *We define an implicit function which translates function symbols into clause symbols. For example, for a symbol function* $\mathtt{f}$, *we denote the associated clause symbol* $\mathtt{f}$. *This function is injective.*

In the following, the term symbols and the clause symbols are denoted by the same symbol when there is no ambiguity.

Definition 24 (Valeurs). *Let* v *a value, we denote* $\mathtt{v}$ *the value in the constraint.*

We can now define the link between the variable environments and the affectations.

Definition 25 (Affectation/Environnement). *Let a variable environment $\mathcal{E}_f$ and a variable translation environment $\mathcal{T}_x$ defined on the variables of $\mathcal{E}_f$. We define the affectation $\mathcal{A}$ corresponding to $\mathcal{E}_f$ modulo the translation of variables denoted $\mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_f$, by:*

$$\mathcal{A} = \{(\mathcal{T}_x(x), \mathcal{E}_f(x)) \mid x \in \mathcal{E}_f\}$$

The expressions The translation rules for expressions are given in Figure 7. They take the following form:

$$\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$$

Such a rule means the expression e is translated into the store σ wrt the environment $\mathcal{T}_x$; the returned value of e is placed in R on the constraints side . The translation rules are the following:

The rule FUNCTION translates a function call $\mathsf{f}(x_1, \ldots, x_n)$. It searches the environment for the variable symbols x_i. The constraint is the call to the clause `f` on these variables.

The rule LET translates a variable binding $\mathbf{let}\, x = e_1\ \mathbf{in}\ e_2$. It translates the expression e_1 with X as the returned variable. Then the expression e_2 is translated wrt the environment $\mathcal{T}_x$ where x is translated in X.

The rule VAR translates a variable. The corresponding constraint imposes the returned variable and the expression are equal.

The rule VALUE is similar to VAR in the case the expression is a value.

The rule IF translates a conditional expression. It translates both expressions e_1 and e_2 to the respective sets of constraints σ_1 and σ_2 wrt the original environment. The entire expression is translated into the constraint $\mathtt{ite}(X, \sigma_1, \sigma_2)$ where X is the constraint variable associated to x.

The rules MATCH and MATCHCATCH translate a pattern matching respectively with and without a catch-all pattern. It consists in translating firstly the set of expressions e_i, associated to the patterns, to a set of constraints system σ_i. The patterns are also translated to a constraint and the constraint variable X corresponding to the matched variable x. The final constraint is a call to the constraint `match` applied to X and $\mathtt{pattern}(X =_h \mathcal{T}_p\, pat_i, \sigma_i)$. The last argument is `fail` when we doesn't have a catch-all pattern and σ_{n_1} otherwise.

Theorem 9. *Let an expression e, a variable translation environment $\mathcal{T}_x$, a variable R and a constraint store σ such as $\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$.*
If we have two affectations $\mathcal{A}_1$, $\mathcal{A}_2$ and a clause environment such as $\mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}_2$ then it exists a value $\mathtt{v}$ such as $\mathcal{A}_2(R) = v$.

Proof. We know by theorem 6 $\mathcal{A}_2$ is total. We should now prove the variable R is defined by $\mathcal{A}_2$. We show R is in the store σ by structural induction on e. Because $\mathcal{A}_2$ is total, we conclude R is defined by $\mathcal{A}_2$.

$$\frac{\mathcal{T}_x(x_1) = X_1 \qquad \ldots \qquad \mathcal{T}_x(x_n) = X_n}{\mathcal{T}_x; R \vdash_C \mathsf{f}(x_1, \ldots, x_n) \longmapsto \mathtt{f}(R, X_1, \ldots, X_n)} \textsc{Function}$$

$$\frac{\begin{array}{c} X \in \mathcal{F}reshC \qquad \mathcal{T}_x; X \vdash_C e_1 \longmapsto \sigma_1 \\ \mathcal{T}_x \oplus (x \leftarrow X); R \vdash_C e_2 \longmapsto \sigma_2 \end{array}}{\mathcal{T}_x; R \vdash_C \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 \longmapsto \sigma_1 \wedge \sigma_2} \textsc{Let}$$

$$\frac{}{\mathcal{T}_x; R \vdash_C v \longmapsto R \diamond v} \textsc{Value} \qquad \frac{\mathcal{T}_x(x) = X}{\mathcal{T}_x; R \vdash_C x \longmapsto R \diamond X} \textsc{Var}$$

$\diamond \in \{=_{fd}, =_h\}$w.r.t. the type of the value/variable

$$\frac{\begin{array}{c} \mathcal{T}_x(x) = X \qquad \mathcal{T}_x; R \vdash_C e_1 \longmapsto \sigma_1 \\ \mathcal{T}_x; R \vdash_C e_2 \longmapsto \sigma_2 \end{array}}{\mathcal{T}_x; R \vdash_C \mathbf{if}\, x \,\mathbf{then}\, e_1 \,\mathbf{else}\, e_2 \longmapsto \mathtt{ite}(X, \sigma_1, \sigma_2)} \textsc{If}$$

$$\frac{\mathcal{T}_x(x) = X \qquad \mathcal{T}_x; R \vdash_C e_i \longmapsto \sigma_i \qquad \forall i \in [\![1, n]\!]}{\mathcal{T}_x; R \vdash_C \begin{array}{l} \mathbf{match}\ x\ \mathbf{with} \\ \mid pat_1 \rightarrow e_1 \\ \quad \vdots \\ \mid pat_n \rightarrow e_n \end{array} \longmapsto \begin{array}{l} \mathtt{match}(X, [\\ \quad \mathtt{pattern}(X =_h \mathcal{T}_p(pat_1), \sigma_1) \\ \quad \vdots \\ \quad \mathtt{pattern}(X =_h \mathcal{T}_p(pat_1), \sigma_n)], \mathtt{fail}) \end{array}} \textsc{Match}$$

$$\frac{\mathcal{T}_x(x) = X \qquad \mathcal{T}_x; R \vdash_C e_i \longmapsto \sigma_i \qquad \forall i \in [\![1, n+1]\!]}{\mathcal{T}_x; R \vdash_C \begin{array}{l} \mathbf{match}\ x\ \mathbf{with} \\ \mid pat_1 \rightarrow e_1 \\ \quad \vdots \\ \mid pat_n \rightarrow e_n \\ \mid \_ \quad \rightarrow e_{n+1} \end{array} \longmapsto \begin{array}{l} \mathtt{match}(X, [\\ \quad \mathtt{pattern}(X =_h \mathcal{T}_p(pat_1), \sigma_1) \\ \quad \vdots \\ \quad \mathtt{pattern}(X =_h \mathcal{T}_p(pat_1), \sigma_n)\ ], \\ \quad \sigma_{i+1}) \end{array}} \textsc{MatchCatch}$$

Fig. 7. Translation of the MiniFocal expressions into constraint

Theorem 10. *Let two expressions e_1, e_2 and two variables x, R and two stores σ_1, σ_2, a variable translation environment $\mathcal{T}_x$ such as $\mathcal{T}_x; R \vdash_C \textbf{let}\, x = e_1$ **in** $e_2 \longmapsto \sigma_1 \wedge \sigma_2$. The variable X associated to x is not in the store σ_1.*

Proof. This is proved by definition of LET. *When we translate e_1, the variable R is not the returned variable and is not in $\mathcal{T}_x$.*

Function Environments

Definition 26 (Function Environnements). *Let $\mathcal{E}_f$ a function environment. We define $\mathcal{E}_{\mathtt{cl}}$ as the clause environment corresponding to $\mathcal{E}_f$ such as:*

$$\mathcal{E}_{\mathtt{cl}} = \{\mathtt{f} \leftarrow <R, X_1, \ldots, X_n \rightsquigarrow \sigma> \mid \mathcal{E}_f(f) = <x_1, \ldots, x_n \rightsquigarrow e> \wedge \\ (x_1 \leftarrow X_1), \ldots, (x_n \leftarrow X_n); R \vdash_C e \longmapsto \sigma\}$$

5.4 Correctness and completeness of the translation

We have now put the basis for proving the correctness and the completeness of the translation. These two theorems are fundamental because they assure, firstly that every solution of the constraint system corresponds to an evaluation of the expression (from which the constraint system is obtained), secondly when an expression evaluates to a value the resulting constraint system contains a corresponding solution. They both assure the equivalence between the two universes.

Theorem 11 (Correction). *Let an expression e, a constraint system σ and a value v such as:*

$$\mathcal{E}_x; \mathcal{E}_f \vdash_N e \rhd v \\ \mathcal{T}_x; R \vdash_C e \longmapsto \sigma$$

let also an affectation $\mathcal{A}$ such as

$$\{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \\ \mathcal{A}(R) = \mathtt{v}$$

It exists an affectation $\mathcal{A}'$ such as:

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'$$

Proof. By induction on the evaluation of e:

Case NVAR
We have $\sigma = \{R =_{fd} X\}$ or $\sigma = \{R =_h X\}$ with $\mathcal{T}_x(x) = X$, this case is obvious.

Case NENTIER NCONSTRUCTEUR NCONSTANTE
Similar to NVAR.

Case NIf_true NIf_false NFiltrage

Proved by case analysis on the value matched/tested and then by induction hypothesis.

Case NApply

Under the hypothesis:

$$\mathcal{E}_x(x_1) = v_1, \ldots, \mathcal{E}_x(x_n) = v_n \tag{1}$$
$$\mathcal{E}_f(f) = <x'_1, \ldots, x'_n \rightsquigarrow e_f> \tag{2}$$
$$(x'_1, v_1), \ldots, (x'_n, v_n); \mathcal{E}_f \vdash_N e_f \rhd v \tag{3}$$
$$\mathcal{E}_x; \mathcal{E}_f \vdash_N f(x_1, \ldots, x_n) \rhd v \tag{4}$$
$$\mathcal{T}_x; R \vdash_C f(x_1, \ldots, x_n) \longmapsto \sigma \tag{5}$$
$$\{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \tag{6}$$
$$\mathcal{A}(R) = v \tag{7}$$

And with the induction hypothesis:

$$\begin{array}{l} \forall\ \mathcal{T}_x\ R\ \mathcal{A}\ \sigma \\ \quad (x'_1, v_1), \ldots, (x'_n, v_n); \mathcal{E}_f \vdash_N e_f \rhd v \Rightarrow \\ \quad \mathcal{T}_x; R \vdash_C e_f \longmapsto \sigma \Rightarrow \\ \quad \{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} (x'_1, v_1), \ldots, (x'_n, v_n) \Rightarrow \\ \quad \mathcal{A}(R) = v \Rightarrow \\ \qquad\qquad \exists\, \mathcal{A}', \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \end{array} \tag{8}$$

We should prove there exists an affectation $\mathcal{A}'$ such as:

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'$$

By hypothesis 4 we deduce $\sigma = \mathtt{f}(R, X_1, \ldots, X_n)$ with $\mathcal{T}_x(x_1) = X_1$, ..., $\mathcal{T}_x(x_n) = X_n$. The only rule we can apply to obtain this goal is Call.
Thus we have $\mathcal{A} = \mathcal{A}'$ and we should prove there exists a store σ' such as this new goal is verified:

$$\mathcal{E}_{\mathtt{cl}}(\mathtt{f}) = <R', X'_1, \ldots, X'_n \rightsquigarrow \sigma'>$$

This is verified by definition of $\mathcal{E}_{\mathtt{cl}}$. We have new to prove there exists an affectation $\mathcal{A}_2$ such as:

$$(R' \leftarrow \mathcal{A}(R)), (X'_1 \leftarrow \mathcal{A}(X_1)) \ldots, (X'_n \leftarrow \mathcal{A}(X_n)); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma' \longmapsto \mathcal{A}_2$$

The hypothesis 8 proved it. The four requirement are given by hypothesis 3, the definition of $\mathcal{E}_{\mathtt{cl}}$, the definition of $=_{\mathcal{T}_x}$ and by hypothesis 7.

Case NLet

In this case, we have $e = \mathbf{let}\, x = e_1\ \mathbf{in}\ e_2$. By decomposition of e we have the following hypothesis:

$$\mathcal{E}_x; \mathcal{E}_f \vdash_N e_1 \rhd v_1 \tag{9}$$
$$\mathcal{E}_x \oplus (x \leftarrow v_1); \mathcal{E}_f \vdash_N e_2 \rhd v_2 \tag{10}$$
$$\mathcal{T}_x; R \vdash_C \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \longmapsto \sigma \tag{11}$$
$$\mathcal{E}_x; \mathcal{E}_f \vdash_N \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \rhd v_2 \tag{12}$$
$$\{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \tag{13}$$
$$\mathcal{A}(R) = v_2 \tag{14}$$

And the two induction hypothesis:

$$\begin{array}{l} \forall\ \mathcal{T}_x\ \ R\ \ \mathcal{A}\ \ \sigma \\ \quad \mathcal{E}_x; \mathcal{E}_f \vdash_N e_1 \rhd v_1 \Rightarrow \\ \quad \mathcal{T}_x; R \vdash_C e_1 \longmapsto \sigma \Rightarrow \\ \quad \{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \Rightarrow \\ \quad \mathcal{A}(R) = v_1 \Rightarrow \\ \quad \exists \mathcal{A}', \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \end{array} \tag{15}$$

$$\begin{array}{l} \forall\ \mathcal{T}_x\ \ R\ \ \mathcal{A}\ \ \sigma \\ \quad \mathcal{E}_x \oplus (x \leftarrow v_1); \mathcal{E}_f \vdash_N e_2 \rhd v_2 \Rightarrow \\ \quad \mathcal{T}_x; R \vdash_C e_2 \longmapsto \sigma \Rightarrow \\ \quad \{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \oplus (x \leftarrow v_1) \Rightarrow \\ \quad \mathcal{A}(R) = v_2 \Rightarrow \\ \qquad \exists \mathcal{A}', \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \end{array} \tag{16}$$

We have to prove there exists an affectation $\mathcal{A}'$ such as:

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}'$$

By hypothesis 11 we deduce $\sigma = \sigma_1 \wedge \sigma_2$ and the two hypothesis (for a fresh X):

$$\mathcal{T}_x; X \vdash_C e_1 \longmapsto \sigma_1 \tag{17}$$
$$\mathcal{T}_x, (x \leftarrow X); R \vdash_C e_2 \longmapsto \sigma_2 \tag{18}$$

The only applicable rule for proving it is Conjonct2*. Thus, we should find two affectations $\mathcal{A}_1$, $\mathcal{A}'$ such as the two following affirmations are true:*

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1$$
$$\mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}'$$

For the first one, we apply the theorems 7 and 8. It gives $(\{R\} \lhd \mathcal{A}) \oplus (X \leftarrow v_1); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1$ because R is non free on σ_1 and by 17. We prove this new goal by hypothesis 15 together with the hypothesis 9, 17, 13 and by definition of an affectation.

For the second one, we apply the hypothesis 16 with the hypothesis 10, 18, 13 and also 14. We obtain the new goal

$$\mathcal{A} \oplus (X \leftarrow v_1); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}' \tag{19}$$

Except for X, the free variables of σ_1 are also non free in σ_2. By the first goal and from 6 we deduce $\mathcal{A}_1$ is $\mathcal{A}$ in which we add the definitions of σ_1 free variable. With these two facts, we can apply weakening theorem on 19 for proving the goal. As a last remark, when we proved the first goal, we show affecting v_1 to X leads to $\mathcal{A}_1$.

Theorem 12 (Completeness). *Let an expression e and a constraint system σ such as:*

$$\mathcal{T}_x; R \vdash_C e \longmapsto \sigma$$

let also a variable environment $\mathcal{E}_x$ and two affectations $\mathcal{A}$, $\mathcal{A}'$ such as:

$$\begin{array}{l} \{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \\ \mathcal{A}(R) = \mathtt{v} \\ \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \end{array}$$

we have the following fact:

$$\mathcal{E}_x; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v$$

Proof. By induction on the definition of $\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}$:

Case CALL

For the variables $X_1, X_1', \ldots, X_n, X_n'$, a function symbol $\mathtt{f}$, two constraints store σ, σ'. Under the hypothesis:

$$(X_1' \leftarrow \mathcal{A}(X_1)), \ldots, (X_n' \leftarrow \mathcal{A}(X_n)); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \tag{20}$$

$$\begin{array}{l} \forall\ R\ v\ e\ \mathcal{T}_x\ \mathcal{E}_x \\ \quad \mathcal{T}_x; R \vdash_C e \longmapsto \sigma \Rightarrow \\ \quad \{R\} \lhd ((X_1' \leftarrow \mathcal{A}(X_1)), \ldots, (X_n' \leftarrow \mathcal{A}(X_n))) =_{\mathcal{T}_x} \mathcal{E}_x \Rightarrow \\ \quad ((X_1' \leftarrow \mathcal{A}(X_1)), \ldots, (X_n' \leftarrow \mathcal{A}(X_n)))(R) = \mathtt{v} \Rightarrow \\ \quad (X_1' \leftarrow \mathcal{A}(X_1)), \ldots, (X_n' \leftarrow \mathcal{A}(X_n)); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma \longmapsto \mathcal{A}' \Rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathcal{E}_x; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v \end{array} \tag{21}$$

$$\mathcal{E}_{\mathtt{cl}}(\mathtt{f}) = <X_1', \ldots, X_n' \rightsquigarrow \sigma > \Gamma \tag{22}$$

We should prove:

$$\begin{array}{l} \forall\ R\ v\ e\ \mathcal{T}_x\ \mathcal{E}_x \\ \quad \mathcal{T}_x; R \vdash_C e \longmapsto \mathtt{f}(X_1, \ldots, X_n) \Rightarrow \\ \quad \{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \Rightarrow \\ \quad \mathcal{A}(R) = \mathtt{v} \Rightarrow \\ \quad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathtt{f}(X_1, \ldots, X_n) \longmapsto \mathcal{A} \Rightarrow \\ \qquad\qquad\qquad\qquad \mathcal{E}_x; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v \end{array}$$

In others words, for a variable R, a value v, an expression e, a variable translation environment $\mathcal{T}_x$ and a variable environment. With 20, 21 and 22 and also the hypothesis:

$$\mathcal{T}_x; R \vdash_C e \longmapsto \mathtt{f}(X_1, \ldots, X_n) \tag{23}$$
$$\{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \tag{24}$$
$$\mathcal{A}(R) = \mathtt{v} \tag{25}$$
$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \mathtt{f}(X_1, \ldots, X_n) \longmapsto \mathcal{A} \tag{26}$$

We should prove: $\mathcal{E}_x; \mathcal{E}_{\mathsf{f}} \vdash_N e \rhd v$.
With 23 we obtain $e = \mathsf{f}(x_2, \ldots, x_n)$ with $X_i = \mathcal{T}_x(x_i)$ and $2 \leq i \leq n$ and $X_1 = R$. Moreover, by the definition 26 and the hypothesis 22, we obtain the hypothesis:

$$\mathcal{E}_{\mathsf{f}}(\mathsf{f}) = <x'_2, \ldots, x'_n \rightsquigarrow e_f> \tag{27}$$
$$(x'_2 \leftarrow X'_2), \ldots (x'_n \leftarrow X'_n); R \vdash_C e_f \longmapsto \sigma \tag{28}$$
$$R = X'_1 \tag{29}$$

Because $e = \mathsf{f}(x_2, \ldots, x_n)$ the conclusion is obtained by application of the rule NApply*. Thus we should prove:*

$$(x'_2, \mathcal{E}_x(x_2)), \ldots, (x'_n, \mathcal{E}_x(x_n)); \mathcal{E}_{\mathsf{f}} \vdash_N e_f \rhd v$$

This is shown by using the hypothesis 21. The four requirements are easily proven. The first one with 28, the second one with the definition of $=_{\mathcal{T}_x}$ and $\lhd$ and $X = X'_1$, the third one with $X = R$ and the hypothesis 25 and the last one is exactly the hypothesis 20.

Case Conjonct1

This case is impossible because $\vdash_C$ does not permit to obtain stores of the form c, σ.

Cas Conjonct2

In this case, from the hypothesis:

$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \tag{30}$$
$$\begin{array}{l} \forall\ R\ v\ e\ \mathcal{T}_x\ \mathcal{E}_x \\ \quad \mathcal{T}_x; R \vdash_C e \longmapsto \sigma_1 \Rightarrow \\ \quad \{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \Rightarrow \\ \quad \mathcal{A}(R) = \mathtt{v} \Rightarrow \\ \quad \mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \Rightarrow \\ \qquad \mathcal{E}_x; \mathcal{E}_{\mathsf{f}} \vdash_N e \rhd v \end{array} \tag{31}$$
$$\mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}_2 \tag{32}$$

$$\begin{array}{l}\forall\ R\ v\ e\ \ \mathcal{T}_x\ \ \mathcal{E}_x \\ \quad \mathcal{T}_x; R \vdash_C e \longmapsto \sigma_2 \Rightarrow \\ \quad \{R\} \lhd \mathcal{A}_1 =_{\mathcal{T}_x} \mathcal{E}_x \Rightarrow \\ \quad \mathcal{A}_1(R) = \mathtt{v} \Rightarrow \\ \quad \mathcal{A}_1; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_2 \longmapsto \mathcal{A}_2 \Rightarrow \\ \qquad\qquad \mathcal{E}_x; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v\end{array} \tag{33}$$

And the hypothesis:

$$\mathcal{T}_x; R \vdash_C e \longmapsto \sigma_1 \wedge \sigma_2 \tag{34}$$
$$\{R\} \lhd \mathcal{A} =_{\mathcal{T}_x} \mathcal{E}_x \tag{35}$$
$$\mathcal{A}(R) = \mathtt{v} \tag{36}$$
$$\mathcal{A}; \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \wedge \sigma_2 \longmapsto \mathcal{A}_2 \tag{37}$$

We should proved $\mathcal{E}_x; \mathcal{E}_{\mathtt{f}} \vdash_N e \rhd v$.
By hypothesis 34, on a $e = \mathbf{let}\, x = e_1\ \mathbf{in}\ e_2$ *we obtain the hypothesis (with fresh* X*):*

$$\mathcal{T}_x; X \vdash_C e_1 \longmapsto \sigma_1 \tag{38}$$
$$\mathcal{T}_x \oplus (x \leftarrow X); R \vdash_C e_2 \longmapsto \sigma_2 \tag{39}$$

By applying the theorem 9 on the hypotheses 38 and 30, we show there exists a value $\mathtt{v}_1$ *such as* $\mathcal{A}_1(X) = \mathtt{v}_1$. *With the theorem 7 and the hypothesis 30 we deduce:*

$$\mathcal{A} \oplus (X \leftarrow \mathtt{v}_1), \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \tag{40}$$

Moreover, by the theorem 10, the variable R *is not in* σ_1 *and thus we can delete the definition of* R *in* $\mathcal{A}$ *in the last hypothesis:*

$$(\{R\} \lhd \mathcal{A}) \oplus (X \leftarrow \mathtt{v}_1); \mathcal{E}_{\mathtt{cl}} \vdash_S \sigma_1 \longmapsto \mathcal{A}_1 \tag{41}$$

The only rule which permits to evaluate e *is* NLET. *We have to prove then the two following goals:*

$$\begin{array}{l}\mathcal{E}_x; \mathcal{E}_{\mathtt{f}} \vdash_N e_1 \rhd v_1 \\ \mathcal{E}_x, (x \leftarrow v_1); \mathcal{E}_{\mathtt{f}} \vdash_N e_2 \rhd v_2\end{array}$$

The first one can be proved with hypothesis 31. The four requirements of 31 are given by 38, the definition of $\oplus$ *and* $\lhd$ *and the fact* X *is fresh, the hypothesis 35 and the hypothesis 41.*
For proving the second goal, we apply the reinforcement theorem before applying hypothesis 32. Let V *the set of variables defined in* $\mathcal{A}_1$ *and undefined in* $\mathcal{A}$ *(except* X*). We have* $V = \{Y_1, \ldots, Y_n\}$ *and* $\mathcal{A}_1(Y_i) = \mathtt{v}'_i$ *with* $1 \leq i \leq n$.

Let n fresh variables $y_1, \ldots, y_n$. We apply the reinforcement theorem on the goal. We obtain the new goal:

$$\mathcal{E}_x, (x \leftarrow v_1), (y_1 \leftarrow v_1'), \ldots, (y_n \leftarrow v_n); \mathcal{E}_f \vdash_N e_2 \rhd v_2$$

We can apply the hypothesis 31, the four requirements are verified either directly by another hypothesis either the definition of $=_{\mathcal{T}_x}$.

Case EqFdI EqFdX EqFdHC EqFdHX

They are obvious.

Case EqFdOp

By hypothesis, the operator of MiniFocal and the constraint system are equivalents.

Case Ite Match_pat

In both cases, we do a case analysis on the matched/tested variable. We can conclude by applying an induction hypothesis.

6 Implementation and Results

6.1 Implementation

FocalTest takes a MiniFocal program and a (non elementary) property as input. It breaks the property under test into elementary properties and for each of them, it produces positive test data which are then executed. The tool includes a parser, a module that breaks the complex properties into elementary ones, a preprocessor that normalizes the function definitions and the elementary properties, a constraint generator, a constraint library that contains the implementation of the operators and a test harness generator. For constraint reasoning, FocalTest is mainly developed in SICStus Prolog and makes an extensive use of the CLP(FD) library of SICStus Prolog. This library implements several arithmetical constraints as well as several labeling heuristics. Both operators `ite/3` and `match/2` are implemented using its *global constraint interface*. Hence, they are considered exactly as any other constraint of the CLP(FD) library. All our experiments have been computed using a *2.33Ghz clocked Intel Core 2 Duo* with *2Go 667 MHz DDR2 SDRAM*. Numeric values are 16 bits values.

6.2 FocalTest Experimental Evaluation

The first purpose of the evaluation is to compare random test data generation with constraint based generation. Furthermore we want to exercise different definitions of `ite` and `match`. In particular we want to measure the impact of using operators `ite`/`match` w.r.t. a natural and naive Prolog implementation and also the impact of using backward reasoning rules.

Context of the Experiment. We evaluated FocalTest as follows. We took 6 examples (listed below) and asked FocalTest to generate 10 positive test data for each elementary property issued from them. In order to evaluate the semantics of the operators, we have executed the examples with 4 strategies of test cases generation. Three of them use constraint solving and the last one concerns the random generation (where the test data are randomly generated until we obtain 10 positive test data). The three constraints solving strategies differ in the implementation of the `ite` and `match` constraints. `fb` denotes the implementation described in this paper (forward and backward rules), `noback` the implementation of both operators with only the forward reasoning rules, the last constraint strategy, `naive`, implements `ite` and `match` with Prolog choice points. Thus $\texttt{ite}(c, t, e)$ is implemented with $(c, t); (\neg c, e)$ and `match` has a similar semantics.

For all examples and strategies, we measured the time of the 10 test data generation (with the Unix time command). We didn't count the time of constraints generation, the time for producing the test harness, and the execution of the test data. We dropped the trivial test cases, that is, the test data which contain small size data like the empty list or a singleton.

The examples. The 6 performed examples are the following:
`avl` is the implementation of the avl trees. The property under test establishes that the insertion of an element in an avl (of integers) results in an avl:

$$\forall t \in \texttt{tree(int)}, \forall e \in \texttt{int}, \texttt{is\_avl}(t) \rightarrow \texttt{is\_avl}(\texttt{insert\_avl}(e, t))$$

`sorted_list` is the similar property over sorted lists:

$$\forall t \in \texttt{list(int)}, \forall e \in \texttt{int}, \texttt{sorted}(t) \rightarrow \texttt{sorted}(\texttt{insert\_list(e,t)})$$

`min_max` is a basic property about the computation of the minimum and the maximum element of a list:

$$\forall l \in \texttt{list(int)}, \forall min\ max\ e \in \texttt{int}, \texttt{is\_min}(min, l) \rightarrow \texttt{is\_max}(max, l) \rightarrow \\ (\texttt{min\_list}(e :: l) = \texttt{min\_int}(min, e) \wedge \texttt{max\_list}(e :: l) = \texttt{max\_int}(max, e))$$

`sum_list` specifies the sum of the elements of the concatenation of 2 lists:

$$\forall s1, s2 \in \texttt{int}, s1 = \texttt{plus\_list}(l1) \rightarrow s2 = \texttt{plus\_list}(l2) \rightarrow \\ s1 + s2 = \texttt{plus\_list}(\texttt{append}(l1, l2))$$

`triangle` is the classical triangle exposed in [9]. The `triangle` function takes three lengths as inputs and returns a value describing the nature of the triangle formed by the three lengths (e.g. `Equilateral`). The experiment concerns the soundness properties. We only give here the property concerning equilateral triangles, `tri_correct_equi` (the other ones are similar):

$$\forall x, y, z \in \texttt{int}, \texttt{triangle}(x, y, z) = \texttt{Equilateral} \rightarrow x = y \wedge y = z$$

`voter 2-out-of-3` is a component used in the industry for computing a unique value from the data obtained via three sensors [10]. It can compute a value

even if one of the sensors is out of order. The function `vote` takes three integers as inputs and returns a pair composed of an integer and a value in {`Match`, `Nomatch`, `Perfect_match`}. `Match` means that two input values are compatible and then the output is one of them. `Nomatch` means the three inputs are not compatible. `Perfect_match` means the three inputs are compatible. Two integers are said compatible if their difference is less than 10. We take into account the soundness properties of the program, for example (`vote_perfect`):

$$\begin{array}{l}\forall v1, v2, v3 \in \texttt{int},\\ \quad \texttt{compatible}(v1, v2) \rightarrow \texttt{compatible}(v2, v3) \rightarrow \texttt{compatible}(v1, v3) \rightarrow\\ \quad\quad \texttt{compatible}(\texttt{fst}(\texttt{vote}(v1, v2, v3)), v1) \wedge\\ \quad\quad \texttt{snd}(\texttt{vote}(v1, v2, v3)) = \texttt{perfect\_match}\end{array}$$

The four first examples are interesting because they contain recursive functions with heavily use of pattern matching and the combination of structures of concrete types (lists and trees) with numeric values. The two last examples deal with numeric values and nested conditional expressions. Just notice that some properties are elementary, e.g. `sum_list` and some are not, e.g. `vote_perfect`. This property is splitted in two elementary properties (same precondition, the conclusion of the first (resp. second) one is the first (resp. second) conjunct).

6.3 Results

The two last columns on Table 1 present the results obtained with the random based approach, more precisely the time of test data generation and the number of test data generated until 10 positive test data are found. We notice that two properties of `triangle` are easy to test randomly, this is not astonishing since these properties require three lengths that either form a scalene triangle or don't form a triangle. The `sorted_list` example is also tractable with the random approach because we generate relatively small lists (4 elements) and the probability to generate sorted lists in this case is high. We remark that for most of the other examples, the random approach fails in finding any test data. We consider the random approach fails when it generates 10 millions consecutive non positive test data.

The first columns of Table 1 present the time for each constraint strategy of generation. We choose to label the variables by generating firstly the algebraic structure (the shape of the list or the tree) and lastly the numeric values (integers). We can see in the table the test data are generated rapidly with `fb`. For most examples, `fb` and `noback` have comparable times. Nevertheless, on `avl`, `vote_partial_c3` and the property about sorted lists, `noback` is less efficient than `fb`. This tends to confirm the backward reasoning rules are useful for searching the test data and thus accelerate the computation of a solution. The reason is, when we label a variable, some constraints may be violated but this is only detectable if these constraints belong to one of the guards in `ite` or `match`. In the opposite, the backward reasoning rules allow to detect as earlier as possible such violations since the guards contain all the constraints involved

Programs	Properties	fb	noback	naive	random	
					time	nb of valuation generated
avl	*See section 6.2*	864	10,926	†	9,180,235	13,763,724
sorted_list	*See section 6.2*	39	178,335	†	< 10	1,123
min_max	*See section 6.2*	104	314	†	162,349 *	20,000,000
sum_list	*See section 6.2*	34	57	†	96,826 *	10,000,000
Triangle	tri_correct_equi	133	112	115	66,855	10,561,778
	tri_correct_iso	204	170	171	67,001	10,593,996
	tri_correct_scal	233	196	196	< 10	1,042
	tri_correct_err	34	27	29	< 10	12
Voter	vote_perfect	225	94	54	144,891 *	20,000,000
	vote_range_c1	113	74	54	1,693	225,104
	vote_range_c2	110	77	243	1,608	213,720
	vote_range_c3	152	65	54	1,898	251,580
	vote_partial_c1	218	181	752	149,947 *	20,000,000
	vote_partial_c2	278	191	45	148,905 *	20,000,000
	vote_partial_c3	289	219,915	487	149,965 *	20,000,000

(†) the generation stopped because of lack of memory. Prolog is expected to allocate the maximum memory. No positive test data were found.

(*) the generation stopped before the 10 required test data have been found and no positive test data were found.

Table 1. test data generation: 6 examples, 4 strategies

in `ite` and `match`. This is exemplified by the property `vote_partial_c3` which has a precondition containing nested `ite` at depth 3. For the examples which contain few execution paths (thus less concerned by backward reasoning), the `naive` and `fb` strategies give similar results. Indeed in these cases, `naive` creates few choice points which are quickly handled. For the other examples, in particular the examples that use lists and `avl` trees, `naive` fails in finding test data because many choice points are induced.

In conclusion, these experimentations show firstly, that constraints helps to find test data for testing properties. Secondly, the choice of using the operators like `ite` and `match` together with forward and backward reasoning rules speeds up the computation of a solution. A naive implementation of `ite` and `match` (with Prolog choice points) is ineffective when there are numerous execution paths in a program.

7 Related Works

Using constraint solving techniques to generate test cases is not a new idea in model- or code-based testing. Dick and Faivre [11] and Marre [12] were among the first to introduce *Constraint Logic Programming* for generating test cases from specification models such as VDM or algebraic specifications. These seminal works yield the development of GATEL, a tool that generates test cases for reactive programs written in Lustre. In 1998, Gotlieb et al. proposed using constraint filtering techniques to generate test data for the structural coverage of C programs [7]. The InKa tool and more recently the EUCLIDE tool [13] which resulted from these works addressed C programs containing general loops,

pointers and floating-point computations. Legeard and Peureux proposed in [14] to exploit set solving techniques to generate test cases from B models. These ideas were pushed further through the development of the BZ-TT and JML-TT toolset. In 2001, Pretschner developed the model-based test case generator AUTOFOCUS that exploited search strategies within constraint logic programming [15] and recently, Williams introduced dynamic path-oriented test data generation in a tool called PathCrawler [16]. This method was independently discovered by Godefroid and Sen in the DART/CUTE approach [17,18]. For testing functional programs, most approaches derive from the QuickCheck tool [5] which generate test data at random. GAST is a similar implementation for Clean, while EasyCheck implements random test data generation for Curry [19]. The development of SAT-based constraint solver for generating test data from declarative models also yields the development of Kato [20] that optimizes constraint solving with (Alloy) model slicing. Like some of the above tools such as GATEL, AUTOFOCUS or EUCLIDE, FocalTest relies on finite domains constraint solving techniques. But, it has also two main differences with these approaches. Firstly, it is integrated within the correct-by-construction Focal environment, meaning that it shares code and correction properties with Focal. This has the advantage of combining tests and proofs in a single environment which let FocalTest generates test data only for the most suspicious implementations. Secondly, it uses its own operators implementation for generating test data in the presence of conditionals and pattern-matching operations. This allows various deduction rules to be exploited to find test data that satisfy properties. Unlike traditional *generate-and-test* approaches, this has the advantage of exploiting constraints to infer new domain reductions and then helps the process to converge more quickly towards acceptable solutions.

8 Conclusion

The constraint-based approach we followed relieves FocalTest from using inefficient *generate-and-test* approaches to select test data satisfying given preconditions. However, this work must be extended to deal with high-order functions as they form the nectar of functional programming. We plan to follow existing transformational approaches and high-order constraint-based reasoning for that. Furthermore exploring how the constraint model of the overall properties and programs could be used to formally prove the conformance of the program to its specifications needs further investigation. Exploiting constraint solving in software verification is likely to be an emerging topic able to enlight the convergence of proofs and tests.

References

1. Dubois, C., Hardin, T., Donzeau-Gouge, V.: Building certified components within focal. In: Revised Selected Papers from the Fifth Symp. on Trends in Functional Prog., TFP'04. Volume 5. (2006) 33–48

2. The Coq Development Team: Coq, *version 8.1.* Available at `coq.inria.fr` (2006)
3. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In Basin, D., Wolff, B., eds.: Theorem Proving in Higher Order Logics, TPHOL. (2003) 188–203
4. Berghofer, S., Nipkow, T.: Random Testing in Isabelle/HOL. In: Software Engineering and Formal Methods (SEFM). (2004) 230–239
5. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices **35**(9) (2000) 268–279
6. Carlier, M., Dubois, C.: Functional testing in the focal environment. In: Test And Proof, TAP. (April 2008) 84–98
7. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: Int. Symp. on Soft. Testing and Analysis, ISSTA, Clearwater Beach, FL (1998) 53–62
8. Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(fd). Journal of Logic Prog. **37** (1998) 139–164
9. Myers, G.J.: The ART of SOFTWARE TESTING. John Wiley &sons (2004)
10. Ayrault, P., Hardin, T., Pessaux, F.: Development life cycle of critical software under focal. In: Int. Workshop on Harnessing Theories for Tool Support in Software, TTSS, Istanbul, Turkey (2008)
11. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: First Int. Symp. of Formal Methods Europe, FME, London, UK (1993) 268–284
12. Marre, B.: Toward Automatic Test Data Set Selection using Algebraic Specifications and Logic Programming. In K. Furukawa, ed.: Int. Conf. on Logic Programming, ICLP, Paris (1991) 202–219
13. Gotlieb, A.: Euclide: A constraint-based testing platform for critical c programs. In: 2th Int. Conf. on Software Testing, Validation and Verification, ICST, Denver, CO (Apr. 2009)
14. Legeard, B., Peureux, F.: Generation of functional test sequences from B formal specifications - presentation and industrial case-study. In: Proc. of the 16th IEEE Int. Conf. on Automated Software Engineering, ASE, San Diego, USA, IEEE Computer Society Press (Nov. 2001) 377–381
15. Pretschner, A.: Classical search strategies for test case generation with constraint logic programming. In: Formal Approaches to Testing of Soft., FATES. (2001) 47–60
16. Williams, N., Marre, B., Mouy, P., Roger, M.: Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: Dependable Computing, EDCC. (2005) 281–292
17. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: ACM Conf. on Prog. lang. design and implementation, PLDI. (2005) 213–223
18. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: ESEC/FSE-13, ACM Press (2005) 263–272
19. Christiansen, J., Fischer, S.: Easycheck – test data for free. In: 9th Int. Symp. on Func. and Logic Prog, FLOPS. (2008)
20. Uzuncaova, E., Khurshid, S.: Constraint prioritization for efficient analysis of declarative models. In: 15th Int. Symp. on Formal Methods, FM. (2008)