Exact projection functions for floating point number constraints*

Claude MICHEL

Claude.Michel@sophia.inria.fr
équipe COPRIN
I3S - INRIA
INRIA - Sophia Antipolis
2004, route des Lucioles - BP 93
06902 - Sophia-Antipolis cedex - France

Abstract

This paper introduces a new framework for filtering constraints over floating-point numbers. An important application area where such solvers are required is test case generation, i.e., finding input values for which a selected point in a procedure is executed. It has been shown that the latter problem can be handled efficiently by translating a non-trivial imperative program into a CSP over finite domains. However, when these programs contain arithmetic operations that involve floating-point numbers, the challenge is to compute test data that are valid even when the arithmetic operations performed by the program to be tested are unsafe. Filtering techniques used in solvers over $\mathcal{R}$ do not preserve all solutions of contraints over floating point numbers and thus, cannot be used to prune the domains of the variables. We introduce here a *FP*-$2B$-consistency, a specialization of $2B$-consistency, which is safe for constraints over floating point numbers. More precisely, we provide the mathematical framework for the computation of projection functions required to implement *FP*-$2B$-consistency.

1 Introduction

Structural testing is still widely used in the process of software validation [GBM98]. The generation of test data – i.e. finding input values for which a selected point in a procedure is executed – is the bottleneck of this technique. Thus, the automatic test data generation (ATDG) is a real challenge (e.g. [DO93, ZHM97, Got00, NGS00]).

*This work is partially supported by the RNTL project INKA

Constraint techniques provide an effective framework to handle automatic test data generation. It has been shown that the ATDG problem can be handled efficiently by translating a non-trivial imperative program into a CSP [GBR98, Got00]. Inka [Got00] is an automatic test data generator based on constraint systems. It handles a significant subset of the C language but it is not yet able to deal with floating point expressions. Our goal is to extend Inka with floating point numbers. Thus, this paper focuses on the resolution of constraint systems over floating point numbers.

The set of floating point numbers is finite but very large: there are more than 10^8 floating point values[1] between -1.0 and 1.0. So, combinatorial process used by finite constraint techniques are ineffective to handle constraints over the floats. Solvers over reals use local consistency algorithms based upon interval narrowing techniques (e.g. $2B$-consistency [Lho93], *Box*-consistency [VMK97]) to prune the domains. Unfortunately, these local consistency algorithms do not preserve all solutions of constraints over the floats, e.g., any floating-point value in $[-1.77635683940025046e-15, 1.77635683940025046e-15]$ is a solution of the relation $16.1 = 16.1 + x$ over the floats[2] whereas `PROLOG IV` reduces the domain of x to 0.0. We have introduced in [MRL01] *FP-Box*-consistency, a specialization of *Box*-consistency which is conservative over the floats. The main limitation of *FP-Box*-consistency is that, though the bound of the computed interval are conservative approximations of the solutions, they are not solutions.

$2B$-consistency computes intervals the bounds of which are solutions of the constraint. Moreover, when no variable occurs more than once, $2B$-consistency is much more efficient than *Box*-consistency [CDR98, Gra01]. $2B$-consistency fixpoint algorithms are based on projection functions. The point is that inverse projection functions cannot be directly computed for constraints over the floats: e.g. for constraint $16.0 = 16.0 + x$, the inverse projection function is $\mathbf{x} = [16.0, 16.0] - [16.0, 16.0]$ which yields $[0.0, 0.0]$ while any floating point numbers in $[-8.88178419700125232e-16, 1.77635683940025046e-15]$ is a solution[3].

This paper introduces *FP*-$2B$-consistency – a specialization of $2B$-consistency – which is based on exact projections of constraints over floating point numbers. Exact projections of constraints greatly ease the labelling process: e.g., a filtering of constraints $x * x = 2.0, x \leq 0.0$ yields $\emptyset$ with a rounding mode set to *near* and $[1.4142135623730951, 1.4142135623730951]$[4] – the only solution – with a rounding mode set to $-\infty$.

This paper is organized as followed: section 2 recalls some basics about floating point numbers as well as the necessary notations and definitions; section 3 formally defines floating point constraint system and *FP*-$2B$-consistency; the mathematical framework for the computation of projection functions is introduced in section 4. Finally, section 5 discusses some implementation related issues and conclude.

---

[1] In `double`.
[2] with a rounding mode set to *near*
[3] with a rounding mode set to *near*
[4] with a Sparc floating point unit.

2 Notations and definitions

2.1 Notations

We mainly use the notations suggested by Kearfott [Kea96]. Thus, throughout, boldface will denote intervals, lower case will denote scalar quantities, and upper case will denote vectors and sets. Brackets "[.]" will delimit intervals while parentheses "(.)" will delimit vectors. Underscores will denote lower bounds of intervals and overscores will denote upper bounds of intervals.

We will also use the following notations, which are slightly non-standard:

- $\mathcal{R}$ denotes the set of real numbers; $\mathcal{F}_\kappa$ denotes an idealized discret subset of $\mathcal{R}$ (see definition 3);
- v stands for a constant in $\mathcal{F}_\kappa$, v^+ (resp. v^-) corresponds to the smallest (resp. largest) number of $\mathcal{F}_\kappa$ strictly greater (resp. lower) than v;
- $f : \mathcal{R} \rightarrow \mathcal{R}$ denotes a strictly increasing function over $\mathcal{R}$; f^{-1} denotes the inverse function of f (i.e. the function such that $f^{-1}(f(x)) = x$); $\hat{f}_r$ denotes the exactly rounded implementation of f over $\mathcal{F}_\kappa$ computed with rounding mode $r \in \{-\infty, +\infty, 0, \text{near}\}$, and $\hat{f}_r^{-1}$ denotes the exactly rounded implementation of f^{-1} over $\mathcal{F}_\kappa$ computed with rounding mode $r \in \{-\infty, +\infty, 0, \text{near}\}$;
- $c : {\mathcal{F}_\kappa}^n \rightarrow \mathcal{B}ool$ denotes a constraint over the floats; $X(c)$ denotes the variables occurring in constraint c.

2.2 Intervals over the floats

Definition 1 (Interval over $\mathcal{F}_\kappa$). *An interval* $\mathbf{x} = [\underline{\mathbf{x}}, \overline{\mathbf{x}}]$ *with* $\underline{\mathbf{x}} \in \mathcal{F}_\kappa$ *and* $\overline{\mathbf{x}} \in \mathcal{F}_\kappa$ *is the set* $\{x \in \mathcal{F}_\kappa \mid \underline{\mathbf{x}} \leq x \leq \overline{\mathbf{x}}\}$.

The reader should notice that an interval does not represent an infinite and continuous subset of $\mathcal{R}$ but a finite subset of $\mathcal{F}_\kappa$.

In the following, $\mathcal{I}$ will denote the set of intervals, ordered by set inclusion, and $\mathcal{U}(\mathcal{I})$ will denote the set of unions of intervals.

Definition 2 (Set extension).
Let S *be a subset of* $\mathcal{F}_\kappa$. *The* Hull *of* S *—denoted* $\Box S$*— is the smallest interval* $\mathbf{x}$ *such that* $S \subseteq \mathbf{x}$.

2.3 Basics of floating point numbers

Floating point numbers have been introduced to allow to approximate operations over $\mathcal{R}$ with computers. The most widely available standard for floating point computation is the IEEE 754 standard [ANS85]. For the sake of simplicity, we will not consider the floating point numbers defined by the standard but use rather the following idealized set:

Definition 3 (Idealized set of floating point numbers). $\mathcal{F}_\kappa = \{m.2^e \mid m \in Z, e \in Z, 2^{\kappa-1} \leq |m| < 2^\kappa, e \geq e_{min}\} \cup \{0\}$ *is the set of floating point numbers.*

$\mathcal{F}_\kappa$ is an idealized set of floating point numbers where numbers have a κ bit long mantissa. The purpose of $\mathcal{F}_\kappa$ is to avoid to have to deal with some singularities like ∞ arithmetic or signed zero (i.e. -0.0).

Most of the time, the result of an operation over floating point numbers is not a floating point number. In order to close the operations over $\mathcal{F}_\kappa$, a rounding operator must be applied to the result. The rounding operator Θ maps a value of $\mathcal{R}$ to a floating point number. IEEE 754 defines four rounding modes. The first three rounding modes have the following definitions:

- Rounding to $+\infty$ is defined by $\Theta_{+\infty}(x) = min\{y \in \mathcal{F}_\kappa \mid x \leq y\}$,
- Rounding to $-\infty$ is defined by $\Theta_{-\infty}(x) = max\{y \in \mathcal{F}_\kappa \mid x \geq y\}$,
- Rounding to 0 is defined by

$$\Theta_{\text{zero}}(x) = \begin{cases} \Theta_{-\infty}(x) & \text{iff } x \geq 0, \\ \Theta_{+\infty}(x) & \text{iff } x < 0. \end{cases}$$

Rounding to the nearest, Θ_{near}, is more complicate to define. Informally speaking, x is rounded to the nearest floating point number. When x is equally distant from two floating point numbers, then x is rounded to the floating point number whose mantissa least significant bit is *even*. More formally, rounding to *nearest* is defined by:

$$\Theta_{\text{near}}(x) = x_k \text{ with } x_k \in \mathcal{F}_\kappa \text{ and,}$$

$$\begin{cases} mid(x_k^-, x_k) < x < mid(x_k, x_k^+) \text{ or,} \\ x = mid(x_k^-, x_k) \text{ or } x = mid(x_k, x_k^+) \text{ with } even(LSB(mantissa(x_k))), \end{cases}$$

where $mid(a, b)$ is the middle of $[a, b]$.

The IEEE 754 standard defines exactly rounded operations and functions. A function is exactly rounded when its value is computed over $\mathcal{R}$ with an infinite precision before being rounded to a floating point number. Such a function $\hat{f}_r$ where the rounding mode $r \in \{-\infty, +\infty, 0, \text{near}\}$ has the following property: $\hat{f}_r(x) = \Theta_r(f(x))$.

3 Constraints over floating point numbers

3.1 Floating point CSPs

A *FCSP* (floating-point constraint system) is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ defined by:

- a set of *variables* $\mathcal{X} = \{x_1, ..., x_n\}$;

IN (**in** $\mathcal{C}$, **inout** $\mathcal{D}$)
$Q \leftarrow \{< x_i, c_j > \mid c_j \in \mathcal{C} \text{ and } x_i \in X(c_j)\}$
while $Q \neq \emptyset$
  extract $< x_i, c_j >$ from Q
  $\mathcal{D}' \leftarrow \texttt{narrowing}(\mathcal{D}, x_i, c_j)$
  if $\mathcal{D}' \neq \mathcal{D}$ **then**
    $\mathcal{D} \leftarrow \mathcal{D}'$
    $Q \leftarrow Q \cup \{< x_l, c_k > \mid c_k \neq c_j \ \wedge \ (x_l, x_i) \in X(c_k)\}$
  endif
endwhile

Figure 1: *FP*-$2B$-consistency standard interval narrowing algorithm

- a set $\mathcal{D} = \{D_1, ..., D_n\}$ of current *domains* D_i where D_i is a finite set of possible floating-point values for variable x_i;
- a set $\mathcal{C}$ of constraints between the variables.

A **constraint** c on the ordered set of variables $X(c) = (x_1, ..., x_r)$ is a subset $T(c)$ of the Cartesian product $(D_1 \times ... \times D_r)$ that specifies the *allowed* combinations of values for variables $(x_1, ..., x_n)$. A rounding mode $Round(c)$ is associated to every constraint $c \in \mathcal{C}$.

A **solution** of a *FCSP* defined by the 3-uplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is a n-uplet $< v_1, \ldots, v_n >$ of floating-point values such that $\forall c_j \in \mathcal{C}$, $c_j(v_1, ..., v_n)$ holds w.r.t. the rounding mode $Round(c_j)$.

3.2 *FP*-$2B$-consistency

FP-$2B$-consistency is a specialized $2B$-consistency [Lho93] for floating point number constraints. $2B$-consistency is a relaxation of *Arc*-consistency. Roughly speaking, a constraint c is *FP*-$2B$-consistency if for any variable x in c, there exist values in the domains of all other variables which satisfies c when x is set either to $\underline{\mathbf{x}}$ or to $\overline{\mathbf{x}}$.

Definition 4 (*FP*-$2B$-consistency).
Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a FCSP, $c \in \mathcal{C}$, a k-ary constraint, c is FP-$2B$-consistent if, $\forall x_i \in X(c), D_i = \Box\{v_i \in D_i \mid \exists v_1 \in D_1, \ldots, \exists v_{i-1} \in D_{i-1}, \exists v_{i+1} \in D_{i+1}, \ldots, \exists v_k \in D_k$ such that $c(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n)$ holds w.r.t. the rounding mode $Round(c)\}$.
A FCSP is FP-$2B$-consistent, iff all its constraints are FP-$2B$-consistent.

FP-$2B$-consistency is computed by a standard interval narrowing algorithm (see fig. 1) derived from AC3 [Mac77]. The implementation of *FP*-$2B$-consistency is based on projection functions.

For instance, consider constraint $c : \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{x}_3$. Three projection functions can be defined:

$$\begin{cases} f_1 : \mathbf{x}_1 \leftarrow \mathbf{x}_3 - \mathbf{x}_2, \\ f_2 : \mathbf{x}_2 \leftarrow \mathbf{x}_3 - \mathbf{x}_1, \\ f_3 : \mathbf{x}_3 \leftarrow \mathbf{x}_1 + \mathbf{x}_2. \end{cases}$$

Since these functions are monotonic, the bounds of the domains of the variables can be trivially computed with the functions: e.g. the new bound $\underline{\mathbf{x}_1}'$ of variable $\mathbf{x}_1$ is defined by $\underline{\mathbf{x}_1}' \leftarrow min\{f_1(\underline{\mathbf{x}_1}), \underline{\mathbf{x}_1}\}$. More formally, projection functions could be defined in the following way:

Definition 5 (Constraint projection [CDR98]).
$\pi_i(c, \mathcal{X}) : (\mathcal{C}, \mathcal{I}) \to \mathcal{U}(\mathcal{I})$ is the projection of c on x_i iff $\pi_i(c, \mathcal{X}) = \{v_i \in D_i \mid \exists(v1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k) \in D_1 \times \ldots \times D_{i-1} \times D_{i+1} \times \ldots \times D_k$ such that $c(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_k)$ holds w.r.t. $Round(c)\}$.

Definition 6 (Approximation of the projection [CDR98]).
$AP_i(c, \mathcal{X}) : (\mathcal{C}, \mathcal{I}) \to \mathcal{U}(\mathcal{I})$ is an approximation of $\pi_i(c, \mathcal{X})$ iff $AP_i(c, \mathcal{X}) = \Box\pi_i(c, \mathcal{X})$.

The computation of AP_i is difficult in the general case. The usual workaround consists in rewriting each constraint in primitive binary and ternary constraints for which the projection is easy to compute [Lho93]. These functions are straightforward to compute for constraints defined over $\mathcal{R}$. Next section shows how these functions can be defined for constraints over floating point numbers.

4 Computation of the projection functions

For the sake of simplicity and without loss of generality, we restrict ourselves to strictly increasing functions over $\mathcal{R}$. We also assume that these functions are exactly rounded.

4.1 Floating point expressions

Expressions in programming languages are more ruled than constraints. A directed acyclic graph (DAG) is often used to represent them. An expression like $x_1 = x_2 + x_3$ in C states the computation of x_1 given x_2 and x_3. However, especially with floating point numbers, it does not allows to directly compute x_2 or x_3 given the two other values.

In this paper, we assume that constraints over floating point numbers are represented by a DAG as this representation capture the orientation of the control flow. As a consequence, two kinds of projection functions are defined:

- *direct projection function* which computes the projection on the variable which represents the result of the expression in the program.
- *inverse projection function* which computes the projection on any other variable involved in the constraint.

These two types of projection functions require appropriate mathematical tools that are introduced in the next sections.

4.2 Computing the direct projection function

In [MRL01], we have shown how the nice properties of the interval arithmetic could be used to build a safe filtering algorithm for *FCSP*. This algorithm mainly relies on the following property: $\square\{y \in \mathcal{F}_\kappa \mid y = \hat{f}_r(x), x \in \mathbf{x}, r \in \{-\infty, +\infty, 0, \text{near}\}\} = [\hat{f}_{-\infty}(\underline{\mathbf{x}}), \hat{f}_{+\infty}(\overline{\mathbf{x}})]$. In other words, interval arithmetic, when computed with floating point numbers and with outward rounding, preserves the solution over $\mathcal{F}_\kappa$ (provided the computation follows the same operation order than the one specified by the expression over $\mathcal{F}_\kappa$; such an order depends on the language). This result provides a mean to compute a safe direct projection function whatever the rounding mode is. When the rounding is known, a more accurate projection function can be defined (an outward rounding overestimates the exact projection).

A well known property of rounding operator Θ is its monotonicity [KM81]: let $u, v \in \mathcal{R}, u \leq v \Rightarrow \Theta_r(u) \leq \Theta_r(v), \forall r \in \{-\infty, +\infty, 0, \text{near}\}$. Thus, when the rounding mode is known, an exact direct projection function could be computed: let $r \in \{-\infty, +\infty, 0, \text{near}\}$ be a rounding mode, $\square\{y \in \mathcal{F}_\kappa \mid y = \hat{f}_r(x), x \in \mathbf{x}\} = [\hat{f}_r(\underline{\mathbf{x}}), \hat{f}_r(\overline{\mathbf{x}})]$.

4.3 Computing the inverse projection functions

This section addresses the computation of the inverse projection. We assume here that $\hat{f}_r^{-1}$ exist and is exactly rounded. The existence of $\hat{f}_r^{-1}$ ensures that $\hat{f}_r$ is an increasing function over $\mathcal{F}_\kappa$ as a strictly increasing function over $\mathcal{R}$ could be constant over $\mathcal{F}_\kappa$.

For a start, let us consider a simple equation: $\hat{f}_{+\infty}(x) = y_k$ where y_k is a floating point constant. Over $\mathcal{F}_\kappa$, the following proposition holds:

Proposition 1. $\square\{x \in \mathcal{F}_\kappa \mid \hat{f}_{+\infty}(x) = y_k\} = \mathbf{x}$ *with*

$$\mathbf{x} = [\Theta^+(f^{-1}(y_k^-)),\ \Theta_{-\infty}(f^{-1}(y_k))]$$

where

$$\Theta^+(x) = \begin{cases} x^+ & \textit{iff } x \in \mathcal{F}_\kappa, \\ \Theta_{+\infty}(x) & \textit{otherwise.} \end{cases}$$

Proof. Since $\hat{f}_{+\infty}$ is an exactly rounded function, we have:

$$\hat{f}_{+\infty}(x) = y_k \Leftrightarrow \Theta_{+\infty}(f(x)) = y_k.$$

It follows from the definition of rounding to $+\infty$ that:

$$y_k^- < f(x) \leq y_k.$$

Thus, over $\mathcal{R}$, any floating point number x such that $\hat{f}_{+\infty}(x) = y_k$ is bounded by:

$$f^{-1}(y_k^-) < x \leq f^{-1}(y_k) \tag{1}$$

Equation 1 yields a bounding of x by two real values. What we are looking for is the smallest interval $\mathbf{x}$ over $\mathcal{F}_\kappa$ which includes all the solutions of the equation. The upper bound of $\mathbf{x}$ is bounded by $x \leq f^{-1}(y_k)$. Thus, we need to get the biggest floating point number u such that $u \leq f^{-1}(y_k)$. This is nothing but the definition of rounding to $-\infty$.
$\underline{\mathbf{x}}$ is bounded by the lowest floating point number l such that $f^{-1}(y_k^-) < l$. By definition, a rounding to $+\infty$ of $f^{-1}(y_k^-)$ gives

$$\Theta_{+\infty}(f^{-1}(y_k^-)) = min\{v \in \mathcal{F}_\kappa \mid v \geq f^{-1}(y_k^-)\}$$

which provides a lower bound of $\mathbf{x}$ such that $\mathbf{x}$ is inclusive of all the solutions of the equation. But we can refine the lower bound to get the smallest $\underline{\mathbf{x}}$ w.r.t. inclusion. The point with the lower bound is that if the result of $f^{-1}(y_k^-)$ is a floating point number, then $\Theta_{+\infty}(f^{-1}(y_k^-)) = f^{-1}(y_k^-)$ while we have a strict inequality. Thus, each time the computation is exact, the lowest floating point number l such that $l > f^{-1}(y_k^-)$ is $l = (\Theta_{+\infty}(f^{-1}(y_k^-)))^+$. □

It is a well known result that rounding operation preserve the monotonicity. The previous formula makes use of the Θ^+ operator which is also a monotonic operator. Intuitively, the Θ^+ just changes the mapping the $\Theta_{+\infty}$ operator used to do from $(y_k^-, y_k] \to y_k$ to $[y_k^-, y_k) \to y_k$. Thus, the proposition 1 could now be extended to intervals, i.e. when the parameter of the function is an interval instead of a single floating point number.

Proposition 2. $\Box\{x \in \mathcal{F}_\kappa \mid \hat{f}_{+\infty}(x) = y, y \in \mathbf{y}\} = \mathbf{x}$ *with*

$$\mathbf{x} = [\Theta^+(f^{-1}(\underline{\mathbf{y}}^-)),\ \Theta_{-\infty}(f^{-1}(\overline{\mathbf{y}}))].$$

Proof. Let $S = \{x \in \mathcal{F}_\kappa \mid \hat{f}_{+\infty}(x) = y, y \in \mathbf{y}\}$. By induction, we have

$$\forall x \in S,\ \underline{\mathbf{y}}^- < f(x) \leq \overline{\mathbf{y}}$$

Thus, over $\mathcal{R}$

$$f^{-1}(\underline{\mathbf{y}}^-) < x \leq f^{-1}(\overline{\mathbf{y}}) \tag{2}$$

Equation 2 is a bounding over $\mathcal{R}$ of the solutions. To get the solutions over $\mathcal{F}_\kappa$, the bounds must be rounded. The biggest floating point number lower than $f^{-1}(\overline{\mathbf{y}})$ is, by definition, $\Theta_{-\infty}(f^{-1}(\overline{\mathbf{y}}))$. The left bound of our set over $\mathcal{F}_\kappa$ is, as in proposition 1, given by $\Theta^+(f^{-1}(\underline{\mathbf{y}}^-))$. □

Figure 2 shows the propositions for the other rounding modes. They can be proved following the same reasoning as for rounding to $+\infty$. Note that the operator Θ^- introduced in figure 2 preserves the monotonicity (intuitively, $\Theta_{-\infty}$ maps $[y_k, y_k^+) \to y_k$ while Θ^- maps $(y_k, y_k^+] \to y_k$).

The next proposition helps to implement the inverse projection and provides a mean to compute a safe inverse projection when the rounding mode is unknown. It is straightforward to show that:

Proposition 6. $\Box\{x \in \mathcal{F}_\kappa \mid \hat{f}_r(x) = y, y \in \mathbf{y}, r \in \{+\infty, -\infty, 0, near\}\} = \mathbf{x}$ *with*

$$\mathbf{x} = [\Theta^+(f^{-1}(\underline{\mathbf{y}}^-)),\ \Theta^-(f^{-1}(\overline{\mathbf{y}}^+))].$$

Proposition 3. $\square\{x \in \mathcal{F}_\kappa \mid \hat{f}_{-\infty}(x) = y, y \in \mathbf{y}\} = \mathbf{x}$ *with*

$$\mathbf{x} = [\Theta_{+\infty}(f^{-1}(\underline{\mathbf{y}})),\ \Theta^-(f^{-1}(\overline{\mathbf{y}}^+))]$$

where

$$\Theta^-(x) = \begin{cases} x^- & \textit{iff } x \in \mathcal{F}_\kappa, \\ \Theta_{-\infty}(x) & \textit{otherwise.} \end{cases}$$

Proposition 4. $\square\{x \in \mathcal{F}_\kappa \mid \hat{f}_0(x) = y, y \in \mathbf{y}\} = \mathbf{x}$ *with*

$$\underline{\mathbf{x}} = \begin{cases} \Theta^+(f^{-1}(\underline{\mathbf{y}}^-)) & \textit{iff } \underline{\mathbf{y}} \leq 0, \\ \Theta_{+\infty}(f^{-1}(\underline{\mathbf{y}})) & \textit{iff } \underline{\mathbf{y}} > 0. \end{cases}$$

$$\overline{\mathbf{x}} = \begin{cases} \Theta_{-\infty}(f^{-1}(\overline{\mathbf{y}})) & \textit{iff } \overline{\mathbf{y}} < 0, \\ \Theta^-(f^{-1}(\overline{\mathbf{y}}^+)) & \textit{iff } \overline{\mathbf{y}} \geq 0. \end{cases}$$

Proposition 5. $\square\{x \in \mathcal{F}_\kappa \mid \hat{f}_{near}(x) = y, y \in \mathbf{y}\} = \mathbf{x}$ *with*

$$\underline{\mathbf{x}} = \begin{cases} \Theta^+(f^{-1}(mid(\underline{\mathbf{y}}^-, \underline{\mathbf{y}}))) & \textit{iff LSB of } \underline{\mathbf{y}} \textit{ is odd,} \\ \Theta_{+\infty}(f^{-1}(mid(\underline{\mathbf{y}}^-, \underline{\mathbf{y}}))) & \textit{iff LSB of } \underline{\mathbf{y}} \textit{ is even} \end{cases}$$

$$\overline{\mathbf{x}} = \begin{cases} \Theta^-(f^{-1}(mid(\overline{\mathbf{y}}, \overline{\mathbf{y}}^+))) & \textit{iff LSB of } \overline{\mathbf{y}} \textit{ is odd,} \\ \Theta_{-\infty}(f^{-1}(mid(\overline{\mathbf{y}}, \overline{\mathbf{y}}^+))) & \textit{iff LSB of } \overline{\mathbf{y}} \textit{ is even.} \end{cases}$$

Figure 2: Inverse projection functions

5 Discussion

Up to now, the computability issue have been discarded. Though the results presented here have been obtained under strong hypotheses, they can be easily extended to fit to the IEEE 754 standard.

Inverse projection functions for rounding to $\{-\infty, +\infty, 0\}$ are computable within $\mathcal{F}_\kappa$ without difficulties. Included in the IEEE 754 standard [ANS85] is a flag which is raised whenever the result of an operation (before the rounding operation) does not belong to $\mathcal{F}_\kappa$. Such an information can be used to implement the Θ^+ and Θ^- operators.

The round to the nearest mode cannot be computed within $\mathcal{F}_\kappa$. The middle of two successive floating point values cannot be represented in $\mathcal{F}_\kappa$. It requires a set of floating point numbers $\mathcal{F}_{\kappa+1}$ whose mantissa has one more bit. Nevertheless, the computation of the inverse projection could be achieved by means of a type which supersets the initial type (e.g. `double` to implement an inverse projection function for `simple`), a library which implements a superset of $\mathcal{F}_\kappa$, or with the help of proposition 6.

An implementation based on a full IEEE 754 standard also requires to handle some singularities like ∞ and signed zero arithmetics as well as $max(\mathcal{F})$ and $min(\mathcal{F})$; especially when rounding to $+\infty$ or $-\infty$ are involved.

We have implemented a prototype version of an *FP*-2*B*-consistency to validate our results. It has also shown that, in term of efficiency, projections over $\mathcal{F}_\kappa$ are competitive with projections over $\mathcal{R}$.

Further work concerns the implementation of a complete solver over floating point numbers using *FP*-2*B*-consistency as well as *FP*-*Box*-consistency [MRL01] to prune better variables with multiple occurrences. The application to automatic test data generation is the next step.

Acknowledgements: Many thanks to Bernard Botella and Arnaud Gotlieb for numerous and enriching discussions on this work. We also gratefully thank Michel Rueher for his constructive remarks, and Gilles Trombettoni for his careful reading of this paper.

References

[ANS85] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[CDR98] H. Collavizza, F. Delobel, and M. Rueher. A note on partial consistencies over continuous domains solving techniques. In *Proc. CP98 (Fourth International Conference on Principles and Practice of Constraint Programming), Pisa, Italy, October 26-30*, 1998.

[DO93] R. A. Demillo and A. J. Offut. Experimental Results from an Automatic Test Case Generator. *Transactions on Software Engineering Methodology*, 2(2):109–175, 1993.

[GBM98] A. Gotlieb, B. Botella, and Rueher M. A clp framework for computing structural test data. In *Proc. ISSTA 98 (Symposium on Software Testing and Analysis)*. ACM SIGSOFT, vol. 2, pp. 53-62, 1998.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *Proc. of the Sigsoft International Symposium on Software Testing and Analysis*, 1998.

[Got00] A. Gotlieb. *Automatic Test Data Generation using Constraint Logic Programming*. PhD thesis, PHD Dissertation (in French), Université de Nice–Sophia Antipolis, January 2000.

[Gra01] Laurent Granvilliers. On the combination of interval constraint solvers. *Reliable Computing*, 7(6):467–483, 2001.

[Kea96] R. Baker Kearfott. *Rigorous Global Search: Continuous Problems.* Number 13 in Nonconvex optimization and its applications. Kluwer

Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands, 1996.

[KM81] U. W. Kulish and W. L. Miranker, editors. *Computer arithmetic in theory and practice.* Computer science and applied mathematics, Academic Press, 1981.

[Lho93] O. Lhomme. Consistency techniques for numeric csps. In *Proceedings of IJCAI'93*, pages 232–238, 1993.

[Mac77] A. Mackworth. Consistency in networks of relations. *Journal of Artificial Intelligence*, pages 8(1):99–118, 1977.

[MRL01] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating point numbers. In *(to appear) Proc. CP2001 (Seventh International Conference on Principles and Practice of Constraint Programming), Paphos, Cyprus, November 26-1*, 2001.

[NGS00] Aditya P. Mathur Neelam Gupta and Mary Lou Soffa. Generating test data for branch coverage. In *Proc. of 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, Grenoble , France, September 2000.

[VMK97] P. Van Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune aprroach. *SIAM Journal*, 34(2), 1997.

[ZHM97] H. ZHU, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *Computing Surveys*, 29(4):366–426, December 1997.