Universität
Bielefeld

**Forschungsberichte der
Technischen Fakultät
Abteilung Informationstechnik**

Basic Semantics for Computer Arithmetic

M. Freericks, A. Fauth, A. Knoll

Report 94-06

Universität Bielefeld • Postfach 10 01 31 • 33501 Bielefeld • FRG

Impressum: Herausgeber:
Robert Giegerich, Alois Knoll, Helge Ritter, Gerhard Sagerer,
Ipke Wachsmuth

Anschrift:
Technische Fakultät der Universität Bielefeld,
Abteilung Informationstechnik, Postfach 10 01 31, 33501 Bielefeld

ISSN 0946-7831

Contents

Chapter 1

Introduction

1.1 Motivation

The arithmetic implemented on "real computers" differs from "mathematical" arithmetic in many respects. Consequences of finite word size, such as rounding, underflow, and overflow, have to be considered. Different hardware platforms support different solutions to these problems; some platforms have unique weaknesses. Even some very basic properties of numeric representations are not fully standardized.

In many programming language descriptions, details such as the exact semantics of primitive data types and operations thereon are left out as either irrelevant or implementation-dependent. Within the typical application domains of "high-level" languages, this may be a reasonable approach – computations can be done with types large and precise enough to provide safety margins which ensure that finite-size effects don't affect the results. In "low-level" applications such as signal processing and bit-true simulation, a precise semantics of the underlying numeric system is however needed.

During the last years, we felt the need for a precise semantic underpinning in three of our projects:

- The ALDiSP project [3][5] was concerned with the specification and implementation of asynchronous real-time signal-processing applications in an applicative framework. ALDiSP is an outgrow of the CADiSP designer tool bench study [7], in which the imperative language ImDiSP was also developed [8]. An understanding of rounding and overflow effects is essential for the specification of many common DSP algorithms, and thus an important issue in DSP language frameworks.
- In the context of the CBC project [2], a retargetable code generation back-end for DSP architectures was developed. This compiler is based on the concept of attributed control/data flow graphs as intermediate representation of the algorithm. It employs "primitive operation" nodes that have to be matched to the hardware operators available in given hardware architecture. When matching n-bit operations to m-bit operators, precise notion of the overflow/carry behaviour of the target hardware is needed.
- The nML machine specification language [1][4] defines hardware architectures by their instruction sets. A behavioural instruction set definition is given by listing an execution semantics for each instruction. nML has to provide adequate primitive operations to model machine operators to such a degree of precision that automatic tools can extract all relevant information (nML descriptions are used to specify the target architectures of the CBC compiler).

Each of these projects needed a bit-true model of primitive numeric data types, and operations defined on them.

1.2 Scope of this Report

This report defines a framework that supports a generic model of numeric representations and operations on them. We try to model all user-relevant notions such as "representation", "ideal" and "concrete" operations, and the treatment of

erroneous situations. Our goal in defining this semantics is to provide a notational tool for our own needs in such areas as compiler transformations, generation of hardware simulators, and conversion between data type representations. One possible direct use of this semantics is to form a basis for a software library to support bit-true simulations.

One of the objectives in providing this semantics is its use as a rule base in type inference systems. This report therefore restricts itself to those numeric types that occur naturally in "statically typed" languages, i.e. languages in which each expression has a type which can be computed at compile-time. "`Symbolic" numeric data-types (e.g., those used in Mathematica or Maple) and dynamic types (e.g., those defined for Common Lisp) are not considered.[1]

This report does not purport to give earth-shattering new insight into numeric types; its purpose is to be a reference so that other texts can employ well-defined types and operations by simply pointing here.

1.3 Types and Operations

At the core of this report is a list of *types*. Each type is characterized by its *bit size*, a *decoding function*, and its *domain*. The domain of a type is the set of values modelled by it, usually a subset of the natural or real numbers, occasionally refined with symbolic values like "∞". The decoding function projects bit strings into the domain; an explicit encoding function need only be specified if the encoding is non-unique.

Using this type description, a *standard operator projection* is defined. This projection maps "mathematical operations" onto their "hardware counterparts". The behaviour of the resulting operators can be modified by defining *attributes* that describe

- rounding behaviour,
- overflow behaviour,
- argument sets for which the operator is undefined,
- argument sets for which the operator is unspecified, and
- an error function.

These function- and set-valued attributes can be defined for both operations and types. Precedence rules define how attributes interact.

1.4 Unspecified and Undefined Behaviours

Standards discriminate between *unspecied* and *undefined* behaviour.

An example for unspecified behaviour might be the result of integer division and modulo in `C` when arguments are negative – the result will be well-defined for each particular implementation, but not the same across implementations.

An example for undefined behaviour might be division by zero in `C` – such an operation might cause an exception, or even crash a program. A particular implemention is not required to specify the effects of an undefined operation; the user has to ensue that such an operation does never occur.

Our semantics provides facilities to model both unspecified and undefined behaviour: the primitives provided by our semantics have a very restricted argument range, even if "reasonable" definitions for wider argument ranges exists. For example, the integer division and remainder operators are unspecified on negative arguments, and division by zero is undefined. On top of these "weak" operations, more "powerful" operations can be built.

In some cases, it is necessary to make a function "less defined". For example, the floating-point multiplication and division operations on certain machines have an error of up to three ULPs. To model such behaviour, it is possible to provide an *error function* that determines the number of bits of lost precision.

[1] Mathematica employs symbolic expression values and computes the numeric approximation of such a value only when the symbolic representation becomes too unwieldy, or a numeric result is explicitly requested by the user. In Common LISP, the run-time type of an object might be data-dependent, and type correctness cannot be guaranteed at compile time; it is not even guaranteed that a given value is always of a numeric type.

1.5 Overview

Chapter 2 introduces the basic notions of *object*, *value*, *representation*, and *type*. Some lemmata are defined so as to show the usefulness and completeness of the definitions. Based on these definitions, operations to *cast* and *coerce* values to types are introduced. In our framework, these operations are rather basic and pop out naturally from the definitions of types and objects.

Chapter 3 defines a set of attributes for rounding, overflow, and underflow.

Chapter 4 defines a list of standard operations defined on (nearly) all types.

Chapter 5 defines a standard catalogue of *integer* and *floating point* types.

Chapter 2

Basic Notions

A semantics for numerics is concerned with the description of *numeric types* and the *primitive functions* defined on them. A type describes a *domain*, which is a set of *values*, and the representation of these values by *bit strings*. Values are usually described in terms of "standard" mathematical domains, such as the natural or real numbers. Conversely, a type can be seen to provide an *interpretation* for all bit patterns of a certain size. Such an interpreted bit pattern is called a *numeric object* (or *object*). In other words, a type describes a set of objects. All primitive functions are be defined in such a way that the result type of a primitive can be statically deduced from the argument type. When used for a programming language, our semantics can therefore be used in combination with a static type system. Most types are *parameterized instances* of *generic types*. For example, the generic type `Int` has instances such as `Int(12)` or `Int(37)`.

2.1 Domains

The basic domains of interest are `Num`, `Obj`, `Type`, and `Bitstring`. `Num`is the domain of *mathematical numbers*, and contains several subsets of interest, such as the cardinal numbers, the integers, the real numbers, some symbolic values (or *non-numbers*), and the special value $\perp$(*bottom*) that denotes an undefined result:

$$\begin{aligned}\texttt{Num} &\supseteq \mathbb{N} \cup \mathbb{Z} \cup \mathbb{R} \cup \texttt{NoN} \cup \{\perp\}\\ \texttt{NoN} &\supseteq \{-0, +\infty, -\infty, \texttt{NaN}\}\end{aligned}$$

A few convenience functions can be defined using these subdomain: `isnum(`x`)` determines whether x is a true number; `isnon(`x`)` determines whether it is one of the `NoN`-numbers, `isdef(`x`)` determines whether it is defined.

$$\begin{aligned}&\texttt{isnum}() : \texttt{Num} \to \{t, f\}\\ &\texttt{isnon}() : \texttt{Num} \to \{t, f\}\\ &\texttt{isdef}() : \texttt{Num} \to \{t, f\}\\ &\texttt{isdef}(x) = x \neq \perp\\ &\texttt{isnon}(x) = x \in \texttt{NoN}\\ &\texttt{isnum}(x) = \texttt{isdef}(x) \wedge x \in \texttt{Num} \wedge \not\texttt{isnon}(x)\end{aligned}$$

`Obj` is the domain of computer-encoded numbers, or *numeric objects*. An object consists of two components, a *type* (the internals of which will be explained later) and a *bit string*, which is a sequence of binary digits. A bit string (and, transitively, each object) has a *size*, which is the number of bits needed to represent it. To keep the semantics simple, there is a maximum size ($\mathtt{size}_{\mathtt{max}}$) which has an appropriately large value. There are a number of convenient operators to work with objects, types, bit strings, and sizes:

$$
\begin{aligned}
&\mathtt{Obj} = \mathtt{Type} \times \mathtt{Bitstring} \\
&\mathtt{Size}() = [1, \ldots, \mathtt{size}_{\mathtt{max}}] \\
&\mathtt{Bitstring} = \mathtt{Bit}^1 \cup \mathtt{Bit}^2 \cup \ldots \cup \mathtt{Bit}^{\mathtt{size}_{\mathtt{max}}} \\
&\mathtt{Bit} = \{0, 1\} \\
&\mathtt{sizeof} : \mathtt{Bitstring} \to \mathtt{Size}() \\
&\mathtt{sizeof} : \mathtt{Obj} \to \mathtt{Size}() \\
&\mathtt{sizeof} : \mathtt{Type} \to \mathtt{Size}() \\
&\mathtt{typeof} : \mathtt{Obj} \to \mathtt{Type} \\
&\mathtt{bits} : \mathtt{Obj} \to \mathtt{Bitstring}
\end{aligned}
$$

2.2 Encode and Decode

Each type T has two associated functions, $\mathtt{encode}_\mathrm{T}$ and $\mathtt{decode}_\mathrm{T}$, that mediate between the domains `Bitstring` and `Num`. Encoding and decoding are partial functions; they are only defined on subdomains of `Obj` and `Num`. `decode` (without a subscript) can be applied to any object; the type of the object provides the information as to what decoding method is to be used:

$$
\begin{aligned}
&\mathtt{encode}_\mathrm{T} : \mathtt{Num}_T \to \mathtt{Obj} \\
&\mathtt{decode}_\mathrm{T} : \mathtt{Bitstring}_T \to \mathtt{Num} \\
&\mathtt{decode} : \mathtt{Obj} \to \mathtt{Num} \\
&\mathtt{decode}(x) = \mathtt{decode}_{\mathtt{typeof\,(x)}}(\mathtt{bits}(x)) \\
&\mathtt{Bitstring}_T = \mathtt{Bit}^{\mathtt{sizeof}(T)} \\
&\mathtt{Obj}_T = \{(T, x) \mid x \in \mathtt{Bitstring}_T\} \\
&\mathtt{Num}_T = \mathtt{Domain}(T) = \mathtt{decode}(\mathtt{Obj}_T) = \{v \mid v = \mathtt{decode}_\mathrm{T}(x), x \in \mathtt{Bitstring}_T\}
\end{aligned}
$$

The *domain* of a type T is the set $\mathtt{Num}_T$ of all values from `Num` that can be represented as objects of type T (elements of $\mathtt{Obj}_T$).

All encode/decode functions must fulfill the following equality:

$$\forall T \in \mathtt{Type} : \forall x \in \mathtt{Num}_T : \mathtt{decode}_\mathrm{T}(\mathtt{encode}_\mathrm{T}(x)) = x$$

That is, the decoding is the inverse of the encoding function. There is no such guarantee for

$$\forall s \in \texttt{Bitstring}_T : \texttt{encode}_\texttt{T}(\texttt{decode}_\texttt{T}(s)) = s$$

since it is possible that an encoding is redundant, i.e. that one value is encoded by more than one binary representation.

An encoding (and its type) is *redundant* if there exists values that are represented by more than one bit pattern. A non-redundant encoding is also called *unique*.

$$\forall T \in \texttt{Type} : \texttt{redundant}(T) \equiv \exists s_1, s_2 \in \texttt{Bitstring}_T : s_1 \neq s_2 \wedge \texttt{decode}_\texttt{T}(s_1) = \texttt{decode}_\texttt{T}(s_2)$$
$$\forall T \in \texttt{Type} : \texttt{unique}(T) \equiv \neg\texttt{redundant}(T)$$

2.3 Casting and Coercing

The most basic semantic functions are related to modifications of the type of an object. When an objects's type is changed, its bit string may be left unchanged, or it might be changed as well. Merely replacing the type is performed by the operation `cast`. A `cast` can thus modify the decoded value of an object; its bit-string is re-interpreted. A re-typing with modification of the value is performed by the operation `coerce`:

$$\begin{aligned}
&\texttt{cast} : \texttt{Type} \times \texttt{Obj} \to \texttt{Obj}\\
&\texttt{coerce} : \texttt{Type} \times \texttt{Obj} \to \texttt{Obj}\\
&\texttt{cast}(t, o) = (t, \texttt{bits}(()o))\\
&\texttt{coerce}(t, o) = \texttt{encode}_\texttt{t}(\texttt{decode}(o))
\end{aligned}$$

Note that $\texttt{coerce}(t, o)$ is undefined on objects o that cannot be represented in type t, and that $\texttt{cast}(t, o)$ is undefined when $\texttt{sizeof}(o) \neq \texttt{sizeof}(t)$.

Some obvious lemmata about the `cast` function are:

$$\begin{aligned}
&\forall T_1, T_2, T_3 \in \texttt{Type}, \Rightarrow \forall x \in \texttt{Obj}_{T_3} : \texttt{cast}(T_1, \texttt{cast}(T_2, x)) = \texttt{cast}(T_1, x)\\
&\forall T_1, T_2 \in \texttt{Type}, \Rightarrow \forall x \in \texttt{Obj}_{T_1} : \texttt{cast}(T_1, \texttt{cast}(T_2, x)) = x
\end{aligned}$$

Proof: by definition.

2.4 Projecting Mathematical Operators to Primitives

A standard mathematical operator p_n (of arity n) is easily mapped to its "primitive" counterparts on a type T:

$$\begin{aligned}
\texttt{mapping}_{\text{trivial}}(T, p_n) \quad = \quad & \lambda(x_1, \ldots, x_n).\\
& \texttt{encode}_\texttt{T}(p_n(\texttt{decode}(x_1), \ldots, \texttt{decode}(x_n)))
\end{aligned}$$

This trivial mapping is only possible if the results of p_n can be represented as T-objects, and if p_n is defined on all n-tuples of T. In the more general case, two auxiliary functions are needed to resolve the problems:

$$
\begin{aligned}
\texttt{mapping}_{\text{standard}}(T, p_n) \quad = \quad & \lambda(x_1, \ldots, x_n). \\
& \quad \texttt{encode}_T(r_e), if\, r_e \in \texttt{Num}_T \wedge r_e \neq \bot \\
& \quad \texttt{undef}(T, x_1, \ldots, x_n), if\, r_e = \bot \\
& \quad \texttt{round}(T, v_1, r_e, v_2), if\, \exists v_1, v_2 \in \texttt{Num}_T, v_1 \leq r_e \leq v_2 \\
& \quad \texttt{bound}(T, r_e), otherwise \\
& where \\
& \quad r_e = r, if\, \texttt{error} = \bot \\
& \quad r_e = \texttt{error}(T, r), otherwise \\
& \quad r = p_n(\texttt{decode}(x_1), \ldots, \texttt{decode}(x_n))
\end{aligned}
$$

The function `undef` encapsulates the treatment of undefined argument sets. The function `round` defines the rounding mechanism, i.e. those cases where the true value of p_n lies between to representable values. The function `bound` is called when the result cannot be represented *and* does not lie between to values, i.e. when the value exceeds the bounds (the largest positive/negatuve value that can be represented) of the type. The `error` function, if defined, introduces an error.

The definitions of these functions are provided as *attributes* to either the mapping function, the type argument T, or the arguments $p_1, \ldots p_n$. Per default, none of these functions is defined (or rather, they are defined as $\bot$, i.e. as functions being undefined on all arguments).

When more than one attribute definition is provided, the mapping function's attribute overrides the type's attribute, which overrides any argument's attribute. If only the arguments are attributed, and the attributes are in conflict (i.e., not identical or bottom), this constitutes a type error.

Chapter 3

Attributes

By defining attributes that define `round`, `undef`, and `bound` functions, types can be modified. Such a modified (or *refined*) type is specified by adding the attributes to type's name, as in

$$\texttt{int}(16)\{\texttt{round} = \texttt{nearest}, \texttt{bound} = \texttt{cutoff}\}$$

which describes a common integer type as it might be used for a `C` implementation. Note that only two of the attributes are given, the third (`undef`) is left undefined.

One of the hairiest aspects of numeric representation, the handling of non-numbers (NaN, the infinities) is excluded from the generic type attributes and instead specified via an `undef` attribute specific to each operation.

3.1 Rounding Methods

There are four common[1] rounding methods:

- `nearest` rounds to the nearest representable number; if two numbers are equally near, the one with a zero bit in its last position is chosen,
- `to`$+\infty$ rounds to the next larger number, i.e. towards $+\infty$
- `to`$-\infty$ rounds to the next smaller number, i.e. towards $-\infty$
- `to`$\pm\infty$ rounds towards the ∞ with the same sign as the rounded number
- `tozero` rounds towards zero

Their definitions are are:

3.1.1 `nearest`

$$\begin{aligned}\texttt{nearest}(T, v_l, v, v_u) \quad = \quad & \texttt{encode}_{\mathbf{T}}(v_l), if\, v - v_l < v_u - v \\ & \texttt{encode}_{\mathbf{T}}(v_r), if\, v - v_l > v_u - v \\ & \texttt{encode}_{\mathbf{T}}(v_l), if\, v - v_l = v_u - v \wedge \texttt{encode}_{\mathbf{T}}(v_l)_0 = 0 \\ & \texttt{encode}_{\mathbf{T}}(v_l), if\, v - v_l = v_u - v \wedge \texttt{encode}_{\mathbf{T}}(v_r)_0 = 0\end{aligned}$$

The four arguments to a `round` function are the type in which the result is to be encoded, the largest encodable number smaller than the actual value (i.e., the "value to the left"), the actual value, and the smallest encodable number larger than the actual value (i.e., the "value to the right"). The expression x_i denotes the i-th bit of a bit string x.

[1] All of the presented rounding modes, except `touinf`, are defined in the IEEE-754 standard.

3.1.2 to$+\infty$

$$\texttt{to}{+}\infty(T, v_l, v, v_u) \quad = \quad \texttt{encode}_{\mathbf{T}}(v_u)$$

3.1.3 to$-\infty$

$$\texttt{to}{-}\infty(T, v_l, v, v_u) \quad = \quad \texttt{encode}_{\mathbf{T}}(v_l)$$

3.1.4 to$\pm\infty$

$$\begin{aligned} \texttt{to}{\pm}\infty(T, v_l, v, v_u) \quad = \quad & \texttt{encode}_{\mathbf{T}}(v_u), if\, v \geq 0 \\ & \texttt{encode}_{\mathbf{T}}(v_l), if\, v < 0 \end{aligned}$$

3.1.5 tozero

$$\begin{aligned} \texttt{tozero}(T, v_l, v, v_u) \quad = \quad & \texttt{encode}_{\mathbf{T}}(v_l), if\, v \geq 0 \\ & \texttt{encode}_{\mathbf{T}}(v_u), if\, v < 0 \end{aligned}$$

3.2 Bounding Methods

The bounding methods presented here are

- `cutoff` cuts of the "bits at the top". This is the usual bounding method for integers in high-level programming languages that don't feature bignums.
- `stick(`x, y`)` replaces any result exceeding the bounds with fixed values. This is the method used in IEEE floating point numbers, with the values $+\infty$ and $-\infty$. In fixed point systems, this is called "saturation arithmetic", and the values are MININT and MAXINT.
- `cutoff(`n`)` is a mixture between `cutoff` and `stick(`$., T$`)` he highest n bits are replaced by all-zero or all-one (depending on the sign), the rest bits are left as they are. This method is sometimes used in DSP hardware.[2]

The bounding methods `cutoff` and `cutoff(`n`)` are most easily described by using a "scratchpad type" that is large enough to hold the intermediate results. The size of the scratchpad type depends upon what operations are needed, e.g. for addition/substraction, only one bit more is needed, while for multiplication, the added sizes of the argument types are required.

3.2.1 cutoff

$$\begin{aligned} \texttt{cutoff}(T_{scratch})(T, v) \quad = \quad & (s_{(n-1)} s_{(n-2)} \ldots s_1 s_0) \\ & where \\ & \quad n = \texttt{sizeof}(T) \\ & \quad m = \texttt{sizeof}(T_{scratch}), m \geq n \\ & \quad s = (s_{(m-1)} s_{(n-2)} \ldots s_1 s_0) = \texttt{encode}_{T_{scratch}}(v) \end{aligned}$$

It is assumed that

$$v \in \texttt{Domain}(T_{scratch})$$

.

[2] It is one of the rounding methods provided by SILAGE.

3.2.2 cutoff(n)

$$\begin{aligned}\texttt{cutoff}(T_{scratch})(N)(T,v) \quad = \quad & (p_1 p_2 \ldots p_n s_{(n-N-1)} s_{(n-N-2)} \ldots s_1 s_0) \\ & where \\ & \quad n = \texttt{sizeof}(T) \\ & \quad p_i = 1, if\, v < 0 \\ & \quad p_i = 0, if\, v \geq 0 \\ & \quad m = \texttt{sizeof}(T_{scratch}), m \geq n \\ & \quad s = (s_{(m-1)} s_{(n-2)} \ldots s_1 s_0) = \texttt{encode}_{T_{scratch}}(v)\end{aligned}$$

3.2.3 stick(x, y)

$$\begin{aligned}\texttt{stick}(x,y)(T,v) \quad = \quad & x, if\, v < \texttt{Max}(\texttt{Domain}(T)) \\ & y, if\, v > \texttt{Max}(\texttt{Domain}(T))\end{aligned}$$

Chapter 4

Operations

The standard operations defined here in some variations are

- *addition* and *subtraction*
- *negation*
- *multiplication*
- *division* (for fp numbers)
- *quotiont* (division on non-fp numbers)
- *remainder*
- *shift* on non-fp numbers
- *rotation* on non-fp numbers

4.1 Addition/Subtraction

The basic add/subtract operators are defined via the standard mapping:

$$\begin{aligned} \texttt{add}_{\text{T}} &= \texttt{mapping}_{\text{standard}}(T, \lambda(x, y).x + y) \\ \texttt{sub}_{\text{T}} &= \texttt{mapping}_{\text{standard}}(T, \lambda(x, y).x + y) \end{aligned}$$

The basic operators are undefined on overflow; the most common variants for integer types handle overflow via cutoff:

$$\begin{aligned} \texttt{add}_{\text{T(size)/cutoff}} &= \texttt{add}_{\text{T(size)}}\{bound = \texttt{cutoff}(T(size + 1))\} \\ \texttt{sub}_{\text{T(size)/cutoff}} &= \texttt{sub}_{\text{T(size)}}\{bound = \texttt{cutoff}(T(size + 1))\} \end{aligned}$$

Addition for floating-point numbers needs some extra cases to handle infinities, NaNs and the zeroes:

$$
\begin{array}{rcl}
\texttt{add}_{\text{Float(n,m)}} & = & \texttt{mapping}_{\text{standard}}(Float(n,m), \lambda(x,y).x+y)\{bound=b, round=r, undef=u\} \\
 & where & \\
b & = & \texttt{stick}(+\infty, -\infty) \\
r & = & \texttt{nearest} \\
u(x,y) & = & \texttt{NaN}, if\ x=\texttt{NaN} \vee y=\texttt{NaN} \vee (x=+\infty \wedge y=-\infty) \vee (x=-\infty \wedge y=+\infty) \\
 & & +\infty, if\ x=+\infty \vee y=+\infty \\
 & & -\infty, if\ x=-\infty \vee y=-\infty \\
 & & -0, if\ (x=-0 \wedge y=-0) \vee (x=0 \wedge y=-0) \vee (x=-0 \wedge y=0) \\
 & & y, if\ (x=-0) \\
 & & x, if\ (y=-0)
\end{array}
$$

Subtraction can be defined via addition and negation:

$$\texttt{sub}_{\text{Float(n,m)}}(x,y) = \texttt{add}_{\text{Float(n,m)}}(x, \texttt{neg}_{\text{Float(n,m)}}(y))$$

4.2 Negation

Negation for most types is defined via straighforward subtraction:

$$\texttt{neg}_{\text{T}}(x) = \texttt{sub}_{\text{T}}(\texttt{encode}_{\text{T}}(0), x)$$

Only floating-point types create a small problem, since $0-0=0$, but $\texttt{neg}(0)=-0$. This can be handled by a simple added case:

$$
\begin{array}{rcl}
\texttt{neg}_{\text{Float(n,m)}}(x) & = & \texttt{sub}_{\text{Float(n,m)}}(\texttt{encode}_{\text{Float(n,m)}}(0), x), if\ x \neq 0 \\
 & & \texttt{encode}_{\text{Float(n,m)}}(-0), if\ x=0
\end{array}
$$

4.3 Multiplication

The basic multiplication operator is defined via the standard mapping:

$$\texttt{mul}_{\text{T}} = \texttt{mapping}_{\text{standard}}(T, \lambda(x,y).x*y)$$

$\texttt{mul}_{\text{T}}$is undefined on overflow; the most common variants for integer types handle overflow via cutoff:

$$\texttt{add}_{\text{T(size)/cutoff}} = \texttt{add}_{\text{T(size)}}\{bound=\texttt{cutoff}(T(size+size))\}$$

Multiplication for floating-point numbers needs some extra cases to handle infinities, NaNs and the zeroes:

$$
\begin{aligned}
\texttt{mul}_{\text{Float(n,m)}} &= \texttt{mapping}_{\text{standard}}(Float(n,m), \lambda(x,y).x * y)\{bound = b, round = r, undef = u\} \\
&\quad where \\
b &= \texttt{stick}(+\infty, -\infty) \\
r &= \texttt{nearest} \\
u(x,y) &= \texttt{NaN}, if\ x = \texttt{NaN} \vee y = \texttt{NaN} \\
&= \texttt{NaN}, if\ ((x = +\infty \vee x = -\infty) \wedge (y = 0 \vee y = -0)) \\
&= \texttt{NaN}, if\ ((y = +\infty \vee y = -\infty) \wedge (x = 0 \vee x = -0)) \\
&\quad +\infty, if\ (x = +\infty \wedge (y = +\infty \vee y > 0)) \vee (y = +\infty \wedge (x = +\infty \vee x > 0)) \\
&\quad +\infty, if\ (x = -\infty \wedge (y = -\infty \vee y < 0)) \vee (x = -\infty \wedge (y = -\infty \vee y < 0)) \\
&\quad -\infty, if\ (x = -\infty \wedge (y = +\infty \vee y > 0)) \vee (y = -\infty \wedge (x = +\infty \vee x > 0)) \\
&\quad -\infty, if\ (x = +\infty \wedge (y = -\infty \vee y < 0)) \vee (x = +\infty \wedge (y = -\infty \vee y < 0)) \\
&\quad -0, if\ (x = 0 \wedge y = -0) \vee (x = -0 \wedge y = 0) \\
&\quad y, if\ (x = -0) \\
&\quad x, if\ (y = -0)
\end{aligned}
$$

4.4 Division

"Division" means "floating-point division"; it is sufficiently different from whole-number division to have its own definition.[1]

$$
\begin{aligned}
\texttt{div}_{\text{Float(n,m)}} &= \texttt{mapping}_{\text{standard}}(Float(n,m), \lambda(x,y).x/y)\{bound = b, round = r, undef = u\} \\
&\quad where \\
b &= \texttt{stick}(+\infty, -\infty) \\
r &= \texttt{nearest} \\
u(x,y) &= \texttt{NaN}, if\ x = \texttt{NaN} \vee y = \texttt{NaN} \\
&= \texttt{NaN}, if\ ((x = 0 \vee x = -0) \wedge (y = 0 \vee y = -0)) \\
&= \texttt{NaN}, if\ ((y = +\infty \vee y = -\infty) \wedge (x = +\infty \vee x = -\infty)) \\
&\quad +\infty, if\ (x > 0 \vee x = +\infty) \wedge (y = 0) \\
&\quad +\infty, if\ (x < 0 \vee x = -\infty) \wedge (y = -0) \\
&\quad -\infty, if\ (x > 0 \vee x = +\infty) \wedge (y = -0) \\
&\quad -\infty, if\ (x < 0 \vee x = -\infty) \wedge (y = 0) \\
&\quad 0, if\ x = 0 \wedge (y > 0 \vee y = +\infty) \\
&\quad 0, if\ x = -0 \wedge (y < 0 \vee y = -\infty) \\
&\quad -0, if\ x = 0 \wedge (y < 0 \vee y = -\infty) \\
&\quad -0, if\ x = -0 \wedge (y > 0 \vee y = +\infty)
\end{aligned}
$$

[1] These definitions are horribles examples of special-case-itis; hopefully I didn't forget a case.

4.5 Quotient and Remainder

The integer division operator works together with the remainder operator; its rounding is defind as cutting off everything behind the decimal point (if it were rounding towards the nearest result, negative remainders would ensue!).

$$\texttt{quotient}_{\mathrm{T}} \quad = \quad \texttt{mapping}_{\text{standard}}(T, \lambda(x, y).x/y)\{round = tozero\}$$

The remainder operator is defined via the result of the quotient, multiplication, and subtraction operators:

$$\texttt{remainder}_{\mathrm{T}}(x, y) \quad = \quad \texttt{sub}_{\mathrm{T}}(x, \texttt{mult}_{\mathrm{T}}(y, \texttt{quotient}_{\mathrm{T}}(x, y)))$$

4.6 Shift, Rotate

These operators are defined on the binary image of an operand, not on its mathematical value. They should only be used on non-fp numbers.

$$\begin{aligned}
\texttt{shift/cl}_{\mathrm{T}}((s_{n-1}s_{n-2}\ldots s_1 s_0), c) &= (cs_{n-1}s_{n-2}\ldots s_1)\\
\texttt{shift/cr}_{\mathrm{T}}((s_{n-1}s_{n-2}\ldots s_1 s_0), c) &= (s_{n-2}\ldots s_1 s_0 c)\\
\texttt{shift/l}_{\mathrm{T}}((s_{n-1}s_{n-2}\ldots s_1 s_0)) &= (s_{n-1}s_{n-1}s_{n-2}\ldots s_1)\\
\texttt{shift/r}_{\mathrm{T}}((s_{n-1}s_{n-2}\ldots s_1 s_0)) &= (s_{n-2}\ldots s_1 s_0 s_0)\\
\texttt{rot/cl}_{\mathrm{T}}((s_{n-1}s_{n-2}\ldots s_1 s_0), c) &= ((cs_{n-1}s_{n-2}\ldots s_1), s_0)\\
\texttt{rot/cr}_{\mathrm{T}}((s_{n-1}s_{n-2}\ldots s_1 s_0), c) &= ((s_{n-2}\ldots s_1 s_0 c), s_{n-1})\\
\texttt{rot/l}_{\mathrm{T}}((s_{n-1}s_{n-2}\ldots s_1 s_0), c) &= (s_0 s_{n-1}s_{n-2}\ldots s_1)\\
\texttt{rot/r}_{\mathrm{T}}((s_{n-1}s_{n-2}\ldots s_1 s_0), c) &= (s_{n-2}\ldots s_1 s_0 s_{n-1})
\end{aligned}$$

All shift- and rotate-operators shift by one position; the “c” indicates the presence of a carry bit, the “l/r” distinguish between left and right.

Chapter 5

The Standard Types

Each type will be presented in a uniform, Unix "man(1) page"-like format that comprises the following sections:

Size

$$\texttt{Size}(type) \quad = \quad \text{(a constant that depends on the parameters)}$$

Each instance (value) of a type is represented by a bit string of size $\texttt{Size}(type)$. For most types, any bit string represents a valid object, i.e. there are no invalid bit combinations. Sizes must be ≥ 1; it makes no sense to speak of 0-bit objects.

Domain

$$\texttt{Domain}(type) \quad = \quad \text{(a set of values)}$$

Each type denotes values drawn from a domain $\texttt{Domain}(type)$. This domain is usually subset of the natural or real numbers, but does occasionaly contain "special objects" such as -0, `NaN` ("not a number"), $+\infty$, or $-\infty$. The domain can usually be computed by applying the decoding function to the set of all valid bit strings.

Decoding

$$\texttt{Decode}_{\text{type}} : \{0,1\}^{\texttt{Size}(type)} \rightarrow \texttt{Domain}(type)$$

The decoding function maps bit strings to values. There might also be a encoding function that maps values to bit strings; the encoding function need only be made explicit when more than one value is mapped to one bit string.

Comments

Some comments are made about each type. These regard notable features of the types, similarities and differences with other types, and useful trivia.

5.1 Card(n): n-bit Unsigned Number

Size

$$\texttt{Size}(\text{Card(n)}) \quad = \quad n$$

Domain

$$\texttt{Domain}(\text{Card(n)}) \quad = \quad [0, \ldots, 2^n - 1]$$

Decoding

$$\texttt{Decode}_{\text{Card(n)}}(x_{n-1} \ldots x_0) = \sum_{i=0}^{n-1} 2^i x_i$$

Comments

This is the most "natural" representation of cardinal numbers. Other representations, such as Gray codes (which are based upon the constraing that incrementing a valued changes exactly on bit), are only infrequently employed in common hard- or software systems.

5.2 Int(n): n-bit Signed Number

Size

$$\texttt{Size}(\text{Int(n)}) \quad = \quad n$$

Domain

$$\texttt{Domain}(\text{Int(n)}) \quad = \quad [-2^{n-1}, ..., -1, 0, 1, ..., 2^{n-1} - 1]$$

Decoding

$$\texttt{Decode}_{\text{Int(n)}}(x_{n-1} \dots x_0) = -x_{n-1}2^{n-1} + \sum_{i=1}^{n-1} 2^i x_i$$

Comments

This is the most common representation of integer numbers. The biggest advantage of this representation is the simplicity of addition/subtraction. As a disadvantage, negation must be done by inversion and increment, necessitating a carry step. The domain is not symmetric; therefore the negation function is undefined when applied to MININT (-2^{n-1}).

5.3 Bias(n,b): n-bit Biased Number

Size

$$\texttt{Size}(\text{Bias(n,b)}) \quad = \quad n$$

Domain

$$\texttt{Domain}(\text{Bias(n,b)}) \quad = \quad [-b, \ldots, 2^n - (b+1)]$$

Decoding

$$\begin{aligned} \texttt{Decode}_{\text{Bias(n,b)}}(x) &= \texttt{Decode}_{\text{Card(n)}}(x) - b \\ &= -b + \sum_{i=0}^{n-1} 2^i x_i \end{aligned}$$

Comments

The number b is called the *bias*. This encoding is employed in floating point exponents following IEEE 754. The main advantage of this encoding is that comparing biased numbers can be done with an unsigned comparator, i.e. binary 0 is the smallest number.

5.4 SignedMagnitude(n): n-bit Signed Magnitude Number

Size

$$\texttt{Size}(\text{SignedMagnitude(n)}) \quad = \quad n$$

Domain

$$\texttt{Domain}(\text{SignedMagnitude(n)}) \quad = \quad [-2^{n-1}-1, \ldots -1, -0, +0, 1, \ldots, 2^{n-1}-1]$$

Decoding

$$\begin{aligned} \texttt{Decode}_{\text{SignedMagnitude(n)}}(s\ x_{n-2}\ldots x_0) &= -1^s \texttt{Decode}_{\text{Card(n-1)}}(x_{n-2}\ldots x_0) \\ &= -1^s \sum_{i=0}^{n-2} 2^i x_i \end{aligned}$$

Comments

Signed-magnitude numbers have as their advantages the very cheap negation/absolute operations (flipping/clearing the front bit) and the fact that the domain is symmetric, hence negation is defined on all values.

As a major disadvantage, there are two zeroes. This complicates testing and addition/subtraction. For all operations that return zero, it is unspecified *which* zero is returned. Most implementations treat both zeros as equal, but not identical; the sign() function is a sure way to diffentiate between them, but the result of a simple comparison is undefined.

Signed-magnitude numbers of size < 2 are not very useful.

5.5 UFix(n,m): $n + m$-bit Unsigned Fixed Point Number

Size

$$\texttt{Size}(\text{UFix(n,m)}) \quad = \quad n + m$$

Domain

$$\begin{aligned} \texttt{Domain}(\text{UFix(n,m)}) &= \frac{1}{2^m} Domain_{Card(n+m)} \\ &= [0, \frac{1}{2^m}, ..., \frac{2^n + m - 1}{2^m}] \end{aligned}$$

Decoding

$$\begin{aligned} \texttt{Decode}_{\text{UFix(n,m)}}(x_{(n+m-1)}\ x_{(n+m-2)}\ \cdots\ x_0) &= \frac{1}{2^m} \texttt{Decode}_{\text{Card(n+m)}}(x_{(n+m-1)}\ x_{(n+m-2)}\ \cdots\ x_0) \\ &= \sum_{i=0}^{n+m-1} 2^{i-m} x_i \end{aligned}$$

Comments

Unsigned fixed point number are cardinals scaled by a fixed amount 2^m, i.e., they have m bits behind the "binary point": In a UFix(n,m) number $(x_{n-1} \ldots x_0\, x_{m-1} \ldots x_0)$, bit x_0 has the value 1, and bit x_{m-1} has the value $\frac{1}{2}$.

Ordinary cardinals are just a special case of unsigned fixed point numbers, since

Card(n) = UFix(n,0)

Adding two UFix numbers which have the same scaling factor amounts to adding ordinary cardinals; when fixed point numbers have to be multiplied, the result has to be corrected by re-scaling (i.e., shifting) it m bits right.

5.6 SFix(n,m): $n + m$-bit Signed Fixed Point Number

Size

$$\texttt{Size}(\text{SFix(n,m)}) \quad = \quad n + m$$

Domain

$$\begin{aligned} \texttt{Domain}(\text{SFix(n,m)}) &= \frac{1}{2^m} Domain_{Int(n+m)} \\ &= [\frac{-2^{n+m-1}}{2^m}, ..., -\frac{1}{2^m}, 0, \frac{1}{2^m}, ..., \frac{2^{n+-1}-1}{2^m}] \end{aligned}$$

Decoding

$$\begin{aligned} \texttt{Decode}_{\text{SFix(n,m)}}(x_{n+m-1}\ x_{n+m-2}\ \ldots\ x_0) &= \frac{1}{2^m}\texttt{Decode}_{\text{Int(n+m)}}(x_{n+m-1}\ x_{n+m-2}\ \ldots\ x_0) \\ &= -x_{n-1}2^{n-1} + \sum_{i=1}^{n+m-1} 2^{i-m}x_i \end{aligned}$$

Comments

Signed fixed point number are cardinals scaled by a fixed amount 2^m, i.e., they have m bits behind the "binary point": In a SFix(n,m) number $(x_{n-1}\ldots x_0\ x_{m-1}\ldots x_0)$, bit x_0 has the value 1, and bit x_{m-1} has the value $\frac{1}{2}$.

Ordinary cardinals are just a special case of signed fixed point numbers, since

Int(n) = SFix(n,0)

Adding two SFix numbers which have the same scaling factor amounts to adding ordinary cardinals; when fixed point numbers have to be multiplied, the result has to be corrected by re-scaling (i.e., shifting) it m bits right.

5.7 Float(n,m): IEEE-754 Floating Point Number

Size

$$\texttt{Size}(\text{Float(n,m)}) \quad = \quad n + m$$

Domain

$$\begin{aligned}
\texttt{Domain}(\text{Float(n,m)}) \quad &= \quad \{(-1)^s 2^E F \mid \forall E \in [-(2^{n-1}) + 2, ..., (2^{n-1}) - 1], F \in \texttt{Domain}(Fix(1, m-1))\,; \\
&\cup \quad \{(-1)^s 2^{E\text{min-}1} F \mid \forall F \in \texttt{Domain}(Fix(0, m-1))\,; \\
&\cup \quad \{\texttt{NaN}_{\text{signaling}}, \texttt{NaN}_{\text{quiet}}, +\infty, -\infty\}
\end{aligned}$$

Decoding

$$\begin{aligned}
\texttt{Decode}_{\text{Float(n,m)}}(s\, e_{n-1} \ldots e_0\, f_{m-2} \ldots f_0) \quad &= \\
\text{if } e = 2^n - 1 \wedge f \neq 0 \quad &\text{then} \quad \texttt{NaN} \\
\text{if } e = 2^n - 1 \wedge f = 0 \quad &\text{then} \quad (-1)^s \infty \\
\text{if } 0 < e < 2^n - 1 \quad &\text{then} \quad (-1)^s 2^{e-b} \texttt{Decode}_{\text{UFix(1,m-1)}}(1\, f_{m-2} \ldots f_0) \\
\text{if } e = 0 \wedge f \neq 0 \quad &\text{then} \quad (-1)^s 2^{0-b} \texttt{Decode}_{\text{UFix(1,m-1)}}(0\, f_{m-2} \ldots f_0) \\
\text{if } e = 0 \wedge f = 0 \quad &\text{then} \quad (-1)^s 0 \\
&where \\
e \quad &= \quad \texttt{Decode}_{\text{Card(n-1)}}(e_{n-1} \ldots e_0) \\
f \quad &= \quad \texttt{Decode}_{\text{Card(n-1)}}(f_{m-2} \ldots f_0) \\
b \quad &= \quad 2^{n-1} - 1
\end{aligned}$$

Comments

IEEE-754 defines a number of floating point types (single, single extended, double, double extended) that only differ in their sizes. The type “single” corresponds to Float(24,8); the type “double” corresponds to Float(53,11). “Single extended” is any type of the form Float($\geq 32, \geq 11$), “double extended is any type of the form Float($\geq 64, \geq 15$). The “e”-bits form the exponent, the “f” bits the mantissa. Note that the sign bit is counted as part of the mantissa (The nomenclature with “s”, “e”, and “f” follows the standard document).

The five cases of the decoding function define NaNs, signed infinities, normalized numbers (having an implicit 1 in the first position), denormalized numbers (having an implicit 0 in the first position), and signed zero. (While 99% of all computations is performed on normalized numbers, 95% of all proofs on IEEE numbers consists of getting the other four cases right.)

There are (at least) two NaNs, a *signaling* NaN and a *quiet* NaN. Their exact representation is not specified. Implementations that don't support exceptions can turn signalling NaNs into quiet NaNs. For the purpose of arithmetic analysis, the difference between the NaNs doesn't really matter, since raising an exception amounts to aborting the computation.

Bibliography

[1] A. Fauth, M. Freericks, A. Knoll, *Generation of Hardware Machine Models from Instruction Set Descriptions*, in: VLSI Signal Processing, VI, Eggermont et.al. (eds), IEEE Signal Processing Society, 1993

[2] A. Fauth, A. Knoll, *Automated Generation of DSP Program Development Tools Utilizing a Machine Description Formalism*, Forschungsberichte des Fachbereichs Informatik Nr. 1992/31, Technische Universität Berlin

[3] M. Freericks, A. Knoll, *ALDiSP - eine applikative Programmiersprache für Anwendungen in der digitalen Signalverarbeitung*, Forschungsberichte des Fachbereichs Informatik Nr. 1990/9, Technische Universität Berlin

[4] M. Freericks, *The nML Machine Description Formalism*, Forschungsberichte des Fachbereichs Informatik Nr. 1991/15, Technische Universität Berlin

[5] M. Freericks, A. Knoll, L. Dooley, *The Real-Time Programming Language ALDiSP-0: Informal Introduction and Formal Semantics*, Forschungsberichte des Fachbereichs Informatik Nr. 1992/26, Technische Universität Berlin

[6] Institute of Electrical and Electronics Engineers, Inc., *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, Approved March 21, 1985 (IEEE Standards Board) / Approved July 26, 1985 (American National Standards Institute)

[7] Alois Knoll, Rupert C. Nieberle, *CADiSP – A Graphical Compiler for the Programming of DSP in a Completely Symbolic Way*, Proc. Int. Conf. on Acoustics, Speech and Signal Processing, April 1990

[8] V. Kruckemeyer, A. Knoll, *Eine imperative Sprache zur Programmierung digitaler Signalprozessoren*, Forschungsberichte des Fachbereichs Informatik Nr. 1990/10, Technische Universität Berlin

[9] G.L. Steele, Jr (ed): *Common Lisp - The Language*, Digital Press, 1984

Index