

# Classifying and Formally Verifying Integer Constant Folding

Sabine Glesner<sup>1</sup>

*Fakultät für Informatik  
Universität Karlsruhe  
Karlsruhe, Germany*

Jan Olaf Blech<sup>2</sup>

*Fakultät für Informatik  
Universität Karlsruhe  
Karlsruhe, Germany*

---

## Abstract

Constant folding is a well-known optimization of compilers which evaluates constant expressions already at compile time. Constant folding is valid only if the results computed by the compiler are exactly the same as the results which would be computed at run-time by the target machine arithmetic. We classify different arithmetics by deriving a general condition under which a target-machine arithmetic can be replaced by a compiler arithmetic. Furthermore, we consider integer arithmetics as a special case. They can be described by residue class arithmetics. We show that these arithmetics form a lattice. Using the order relation in this lattice, we establish a necessary and sufficient criterion under which constant folding can be done in a residue class arithmetic that is different from the one of the target machine. Concerning formal verification, we have formalized our proofs in the Isabelle/HOL system. As examples, we discuss the Java and C integer arithmetics and show which compiler arithmetics are valid for constant folding. This discussion reveals also potential sources of incorrect behavior of C compilers.

---

**Keywords:** integer and residue class arithmetic, constant folding, formal correctness, Isabelle/HOL formalization, Java and C integer arithmetic.

---

<sup>1</sup> Email: [glesner@ipd.info.uni-karlsruhe.de](mailto:glesner@ipd.info.uni-karlsruhe.de)

<sup>2</sup> Email: [blech@ipd.info.uni-karlsruhe.de](mailto:blech@ipd.info.uni-karlsruhe.de)

## 1 Introduction

Most programming languages do not specify exactly how arithmetic computations are performed. Instead they refer implicitly to the built-in arithmetic of the processor of the target machine. The advantage is better portability of source programs. Especially integer arithmetic is mostly used for counting tasks in small ranges which behave accurately as the ring  $\mathbb{Z}$  of the integer numbers. The same source program may be translated into machine code for 16- as well as 32- or even 64-bit processors. This situation has implications on the optimizations allowed in a compiler. Constant folding is a well-known optimization in the intermediate representation of compilers which evaluates constant expressions already at compile time. Constant folding is valid only if the results computed by the compiler are exactly the same as the results which would be computed at run-time by the target machine arithmetic. In this paper, we classify integer and floating point arithmetics in a general framework. In particular, we state a sufficient criterion under which a target machine arithmetic may be replaced by a compiler arithmetic. Moreover, we describe sufficient and necessary conditions under which a given integer arithmetic may safely be replaced by some other integer arithmetic. These conditions are efficiently decidable. We discuss the Java and C integer arithmetics and the implications on allowed compiler arithmetics with respect to our criterion of substitutability. Moreover, we present a formal proof of correctness for our criterion, stated in the Isabelle/HOL system.

As a rather amusing motivation for the importance of the compiler arithmetic consider the following presumably true story [Poo94]. In 1994, a major city bank in the UK wanted to find out which of their Pentium processors were afflicted with the Pentium bug. They compiled a test program and checked all their Pentium machines with it. Surprisingly, all of them had the bug. Just to double-check, they also tested their other machines and, even more surprisingly, discovered that they also showed the Pentium bug. After some confusion, they came up with this explanation. The compiler did constant folding, computing the expression intended to reveal the Pentium bug already at compile time. This compiling processor was a buggy Pentium which hardwired the wrong result into the translated machine program. Hence, in turn, the mistake showed up in each run of the program, independently of the arithmetic of the executing processor. This story demonstrates that the arithmetic of the compiler is important and needs to behave exactly as the arithmetic of the target machine in order to guarantee that the optimized target program behaves exactly as the unoptimized target program would do.

With our results in this paper, we introduce a general framework for the substitutability relation between arithmetics. We specialize this general setting for integer arithmetics, yielding an efficiently decidable criterion for substitutability between different integer arithmetics. In section 2, we recall some notations and results from universal and abstract algebra. In section 3, we

present our general framework for the substitutability relation among arithmetics. We consider integer arithmetics in section 4 and show that they can be arranged in a lattice. This lattice is isomorphic to the dual lattice of congruence relations of the ring  $\mathbb{Z}$  of integer numbers. This classification gives us a necessary and sufficient criterion of substitutability between integer arithmetics, expressed by certain intuitive divisor properties. We discuss the integer arithmetics of Java and C in section 5 and show how they can be classified with our schema. This discussion reveals in particular the fundamental difference between Java and C arithmetics. In Java, the results of arithmetic expressions are determined by the Java semantics. In C programs, the results of arithmetic expressions are defined in terms of the target machine arithmetic. In section 6, we describe our Isabelle/HOL formalization. We discuss related work in section 7. In section 8, we conclude with aspects of future work.

## 2 Foundations

Ideally one may think of integer and floating-point arithmetics in programming languages as being equal to the ring  $\mathbb{Z}$  of the integer numbers or to the field  $\mathbb{R}$  of real numbers. Nevertheless, reality is different as nearly all arithmetics in programming languages are implemented by the available arithmetic operations of the target processor. These operations are defined only for a finite number of possible input values and, hence, differ from  $\mathbb{Z}$  or  $\mathbb{R}$ . We describe these processor arithmetics as well as  $\mathbb{Z}$  and  $\mathbb{R}$  as universal algebras. This general setting allows us to view all of these arithmetics in a unified framework. In this section, we recall some standard notations and results from classical and universal algebra. For more details or proofs cf. [Lan79,Ihr88,BS00].

### 2.1 Universal Algebras

**Definition 2.1** [Operations] For each  $n \in \mathbb{N} \cup \{0\}$  and each nonempty set  $\mathcal{A}$ , a mapping  $f : \mathcal{A}^n \rightarrow \mathcal{A}$  is called  $n$ -ary operation on  $\mathcal{A}$ .  $n$  is called the arity of  $f$ . The set of all  $n$ -ary operations on  $\mathcal{A}$  is denoted by  $Op_n(\mathcal{A})$ .  $\diamond$

**Definition 2.2** [Type of Algebras] A type of algebras is a set  $\mathcal{F}$  of *function symbols* such that a nonnegative integer  $n \in \mathbb{N} \cup \{0\}$  is assigned to each function symbol  $f \in \mathcal{F}$ .  $n$  is the *arity* of  $f$ ,  $f$  is an  $n$ -ary function symbol.  $\diamond$

**Definition 2.3** [Universal Algebra] A *universal algebra*  $\mathcal{A}$  of type  $\mathcal{F}$  is an ordered pair  $\mathcal{A} = (A, F)$  where  $A$  is a nonempty set and  $F = (f_{\mathcal{A}} \mid f \in \mathcal{F})$  is a family of operations of finite arity on  $A$ . Thereby an  $n$ -ary operation  $f_{\mathcal{A}} \in Op_n(A)$  is assigned to each  $n$ -ary function symbol  $f \in \mathcal{F}$ .  $A$  is called the *universe* of  $\mathcal{A} = (A, F)$ . The elements in  $F$  are the fundamental operations of  $\mathcal{A}$ . If  $F = \{f_1, \dots, f_k\}$  is finite, we write  $(A, f_1, \dots, f_k)$  for  $(A, F)$ .  $\diamond$

Throughout this paper, we write  $f$  instead of  $f_{\mathcal{A}}$  whenever this simplification is unambiguous.

**Example 2.4** [Groups and Rings] Classical examples of algebras are groups and rings. A *group* is an algebra  $\mathcal{G} = (G, \cdot, ^{-1}, 1)$  in which the following identities are true: (*Abelian groups* satisfy also commutativity:  $x \cdot y = y \cdot x$ .)

$$\begin{aligned} x \cdot (y \cdot z) &= (x \cdot y) \cdot z & (\text{associativity}) & \quad e \cdot x = x \cdot e = x & (\text{neutral element}) \\ x \cdot x^{-1} &= x^{-1} \cdot x = e & (\text{inverse elements}) & \end{aligned}$$

A *ring* is an algebra  $(R, +, -, 0, \cdot)$  where  $+$  and  $\cdot$  are binary operations,  $-$  is a unary and  $0$  is a nullary operation. A ring satisfies the following conditions:  $(R, +, -, 0)$  is an abelian group.  $\cdot$  is an associative operation, and the two distributivity laws are fulfilled:

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z) \quad \text{and} \quad (x + y) \cdot z = (x \cdot z) + (y \cdot z) \quad \blacksquare$$

Given a set of variables  $V$  and a type of algebras  $\mathcal{F}$ , the set of terms  $T(\mathcal{F}, V)$  is defined inductively as usual: All variables  $v \in V$  are terms. If  $f$  is an  $n$ -ary function symbol and  $t_1, \dots, t_n$  are terms, then  $f(t_1, \dots, t_n)$  is a term.

**Definition 2.5** Let  $\mathcal{F}$  be a type of algebras and  $V$  be a set of variables. The term algebra  $\mathcal{T}(V)$  of type  $\mathcal{F}$  over the set of variables  $V$  is the algebra  $\mathcal{T}(V) = (T(\mathcal{F}, V), \mathcal{F})$  such that for each  $f \in \mathcal{F}$ ,  $f_{\mathcal{T}(V)}(t_1, \dots, t_n) = f(t_1, \dots, t_n)$ .  $\diamond$

## 2.2 Homomorphisms

**Definition 2.6** [Homomorphism] Let  $\mathcal{A}$  and  $\mathcal{B}$  be two algebras of the same type  $\mathcal{F}$ . A mapping  $\alpha : A \rightarrow B$  is called a homomorphism from  $\mathcal{A}$  to  $\mathcal{B}$  if  $\alpha(f_{\mathcal{A}}(a_1, \dots, a_n)) = f_{\mathcal{B}}(\alpha(a_1), \dots, \alpha(a_n))$ . The kernel of  $\alpha$ ,  $\ker(\alpha)$ , is defined by  $\ker(\alpha) = \{(a, a') \mid a, a' \in A \wedge \alpha(a) = \alpha(a')\}$ .  $\diamond$

**Theorem 2.7 (Concatenation)** Let  $\alpha : A \rightarrow B$  and  $\beta : B \rightarrow C$  be homomorphisms from  $\mathcal{A}$  to  $\mathcal{B}$  and from  $\mathcal{B}$  to  $\mathcal{C}$ . Then the concatenation of  $\alpha$  and  $\beta$ ,  $\beta \circ \alpha$ , with  $\beta \circ \alpha(a) = \beta(\alpha(a))$  for  $a \in A$ , is also a homomorphism.  $\diamond$

**Example 2.8** [Modulo-Arithmetic  $\mathbb{Z}_n$  of  $\mathbb{Z}$ ] Consider the modulo-arithmetic  $\mathbb{Z}_n$  in  $\mathbb{Z}$ ,  $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$ . Addition is defined as follows:  $x +_{\mathbb{Z}_n} y = (x +_{\mathbb{Z}} y) \bmod n$ . If  $x \in \mathbb{Z}_n$ , then  $-x = n -_{\mathbb{Z}} x$ .  $x -_{\mathbb{Z}_n} y$  is an abbreviation for  $x +_{\mathbb{Z}_n} (-_{\mathbb{Z}_n} y)$ . Multiplication is defined analogously. ( $+_{\mathbb{Z}}$  and  $-_{\mathbb{Z}}$  denote addition and subtraction in  $\mathbb{Z}$ , resp.) It is easy to verify that  $f : \mathbb{Z} \rightarrow \mathbb{Z}_n$  with  $f(x) = x \bmod n$  is a homomorphism from  $\mathbb{Z}$  to  $\mathbb{Z}_n$ .  $\blacksquare$

## 2.3 Congruence Relations and Quotient Algebras

**Definition 2.9** A *congruence relation*  $\theta$  on the algebra  $\mathcal{A} = (A, F)$  is an equivalence relation on  $A$  such that for all  $f \in F$ , if  $f$   $n$ -ary and if  $(t_i, t'_i) \in \theta$ ,  $1 \leq i \leq n$ , then  $(f(t_1, \dots, t_n), f(t'_1, \dots, t'_n)) \in \theta$ . The set of all congruence classes is denoted by  $A/\theta$ , the congruence class containing  $a$  by  $a/\theta$ .  $\diamond$

**Theorem 2.10** Let  $\alpha : A \rightarrow B$  be a homomorphism from  $\mathcal{A}$  to  $\mathcal{B}$ . Then  $\ker(\alpha)$  is a congruence relation on  $A$ .  $\diamond$

**Definition 2.11** [Quotient Algebras] Let  $\mathcal{A} = (A, F)$  be an algebra and  $\theta$  be a congruence relation on  $\mathcal{A}$ . The *quotient algebra* of  $\mathcal{A}$  by  $\theta$ , denoted  $\mathcal{A}/\theta$ , is the algebra whose universe is  $A/\theta$  and whose fundamental operations satisfy:  $f_{\mathcal{A}/\theta}(a_1/\theta, \dots, a_n/\theta) = f_{\mathcal{A}}(a_1, \dots, a_n)/\theta$ .  $\diamond$

**Example 2.12** [Residue Classes of  $\mathbb{Z}$ ] The elements of  $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$  can be viewed as standard representatives of the residue classes  $n\mathbb{Z}+r$ ,  $0 \leq r \leq n-1$

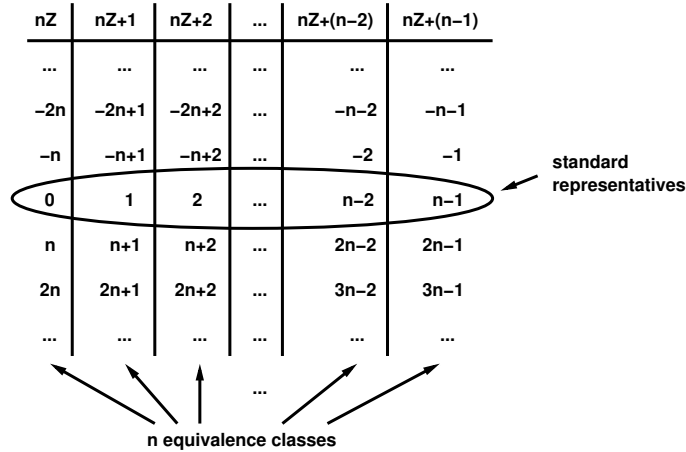


Fig. 1. Residue Classes of  $\mathbb{Z}$

$n-1$ , of  $\mathbb{Z}$ , cf. figure 1. The residue classes are defined as follows (for arbitrary but fixed  $n \geq 1$  and all  $r$  with  $0 \leq r \leq n-1$ ):  $n\mathbb{Z}+r = \{x \in \mathbb{Z} \mid x \bmod n = r\}$ .

These residue classes are the congruence classes of the congruence relation  $\theta_n$  on  $\mathbb{Z}$ :  $x\theta_n y$  iff  $x \bmod n = y \bmod n$ . Clearly,  $\theta_n$  is an equivalence relation on  $\mathbb{Z}$  because it is reflexive, symmetric, and transitive. Moreover, it is a congruence relation because it is closed under the operations  $+\mathbb{Z}_n$ ,  $-\mathbb{Z}_n$ , and  $\cdot\mathbb{Z}_n$ . Hence, we conclude that  $\mathbb{Z}/\theta_n = \mathbb{Z}_n$  is an algebra with well-defined operations as specified in definition 2.11. One can show that  $\mathbb{Z}_n$  is a ring (by Birkhoff’s theorem about equational classes being varieties and vice versa).

Ideals  $J$  of rings  $(R, +, -, 0, \cdot)$  are subsets of  $R$ . In commutative rings, they are defined by three properties:  $0 \in J$ ;  $x, y \in J \Rightarrow x + y \in J$ ; and  $x \in J, a \in R \Rightarrow a \cdot x \in J$ . Each ideal is the kernel of a homomorphism, and, vice versa, each kernel of a homomorphism is an ideal. Furthermore, in  $\mathbb{Z}$ , the ideals are exactly the congruence classes  $n\mathbb{Z}$  for  $n \in \mathbb{N}^+$ . ■

An important result from universal algebra states that the congruence relations of an algebra are a lattice (cf. [BS00]). We use this result when classifying integer arithmetics in section 4.

**Theorem 2.13** *Let  $\mathcal{A}$  be an algebra and let  $\text{Con } \mathcal{A}$  be the set of all congruence relations of  $\mathcal{A}$ . The congruence lattice of  $\mathcal{A}$ , denoted by  $\mathbf{Con } \mathcal{A}$ , is the lattice whose universe is  $\text{Con } \mathcal{A}$  and meets and joins are calculated as follows:*

- $\theta_1 \wedge \theta_2 = \theta_1 \cap \theta_2$

- $\theta_1 \vee \theta_2 = \theta_1 \cup (\theta_1 \circ \theta_2) \cup (\theta_1 \circ \theta_2 \circ \theta_1) \cup (\theta_1 \circ \theta_2 \circ \theta_1 \circ \theta_2) \cup \dots$  or, equivalently,  $(a, b) \in \theta_1 \vee \theta_2$  iff there is a sequence of elements  $a = c_1, c_2, \dots, c_n = b$  such that  $(c_i, c_{i+1}) \in \theta_1$  or  $(c_i, c_{i+1}) \in \theta_2$  for  $1 \leq i \leq n - 1$ .  $\diamond$

### 3 General Classification of Arithmetics

At first sight, one might think that the integer and floating point arithmetics in programming languages are rings and fields, resp. For the modulo-arithmetic on integer numbers, this is true. But already for saturating integer arithmetic, we do not have arithmetic on rings any more. Analogously, floating-point arithmetic behaves like the arithmetic in a field only as long as no rounding errors or overflows occur. In this section, we define the notion of arithmetics and derive the notion of substitutability between algebras. Furthermore, we define substitutability with respect to one specific constant expression.

**Definition 3.1** [Arithmetic] An arithmetic is an algebra  $\mathcal{A}$  of type  $\mathcal{F}$  such that the following function symbols are included in  $\mathcal{F}$ :  $+$ ,  $-$ ,  $\cdot$ ,  $0$ .  $+$  and  $\cdot$  are binary function symbols,  $-$  is a unary and  $0$  is a nullary function symbol.  $\diamond$

This definition captures the classical arithmetics like  $\mathbb{Z}$  and  $\mathbb{R}$  as well as  $\mathbb{Z}_n$ , saturating integer or floating point arithmetics implemented in microprocessors. We do not require any properties of the fundamental operations of the algebra as e.g.  $0 \cdot x = 0$ . Any algebra of the right type is accepted as arithmetic. We need to be able to compare arithmetics. Therefore we define a relation “more precise” which holds for two algebras  $\mathcal{A}$  and  $\mathcal{B}$ , denoted by  $\mathcal{A} \succeq \mathcal{B}$ , if  $\mathcal{A}$  can be used instead of  $\mathcal{B}$  to compute a constant expression. With  $\text{Type}(\mathcal{A})$ , we denote the function symbols which are interpreted by the fundamental operations of the algebra  $\mathcal{A}$ . If  $Op \subseteq \text{Type}(\mathcal{A})$ , then we denote with  $\mathcal{A}|_{Op}$  the following algebra  $(A, \{f_A \mid f \in \text{Type}(\mathcal{A}) \cap Op\})$  which has only a subset of the fundamental operations of  $\mathcal{A}$ , those contained in  $Op$ .

**Definition 3.2** [More Precise  $\succeq$ ] An algebra  $\mathcal{A} = (A, F_A)$  is *more precise* than an algebra  $\mathcal{B} = (B, F_B)$ , denoted by  $\mathcal{A} \succeq \mathcal{B}$ , iff

- $\text{Type}(\mathcal{B}) \subseteq \text{Type}(\mathcal{A})$  and if
- there exists a surjective homomorphism  $f : \mathcal{A}|_{Op(\mathcal{B})} \rightarrow \mathcal{B}$ .  $\diamond$

If  $\mathcal{A} \succeq \mathcal{B}$ , then  $\mathcal{A}$  can be used instead of  $\mathcal{B}$  to evaluate constant expressions. To formalize this, we need a formal definition of constant expressions of an algebra  $\mathcal{A} = (A, F)$ . We define them as the term algebra of the same type over the set of “variables”  $A$  as  $\mathcal{T}(\mathcal{A}) = (T(\text{Type}(\mathcal{A}), A), \text{Type}(\mathcal{A}))$ . If  $f : A \rightarrow B$  is a function defined on  $A$ , then we can lift this function to  $f : \mathcal{T}(\mathcal{A}) \rightarrow \mathcal{T}(\mathcal{B})$  by mapping each  $a \in A$  to  $f(a)$  and each term  $g_A(t_1, \dots, t_n)$  to  $g_B(f(t_1), \dots, f(t_n))$ . Furthermore, we need an evaluation function  $eval_{\mathcal{A}}$  which assigns each term  $t \in T(\text{Type}(\mathcal{A}), A)$  an element of  $A$ .  $eval_{\mathcal{A}} : \mathcal{T}(\mathcal{A}) \rightarrow \mathcal{A}$  is defined inductively as follows: If  $t = a$  for some  $a \in A$ , then  $eval_{\mathcal{A}}(a) = a$ . If  $t = h(t_1, \dots, t_n)$ , then  $eval_{\mathcal{A}}(t) = h_{\mathcal{A}}(eval_{\mathcal{A}}(t_1), \dots, eval_{\mathcal{A}}(t_n))$ .

**Theorem 3.3 (Substitutability)** *Let  $\mathcal{A} = (A, F_A)$ ,  $\mathcal{B} = (B, F_B)$  be algebras,  $\mathcal{A} \succeq \mathcal{B}$ . Let  $f : \mathcal{A}|_{Op(\mathcal{B})} \rightarrow \mathcal{B}$  be the corresp. homomorphism from  $\mathcal{A}|_{Op(\mathcal{B})}$  to  $\mathcal{B}$ . Then there exists a function  $f^{-1} : T(\text{Type}(\mathcal{B}), B) \rightarrow T(\text{Type}(\mathcal{B}), A)$  with  $f(\text{eval}_{\mathcal{A}}(f^{-1}(t))) = \text{eval}_{\mathcal{B}}(t)$  for all  $t \in T(\text{Type}(\mathcal{B}), B)$ .  $\diamond$*

**Proof.** First we define a function  $f^{-1} : \mathcal{B} \rightarrow \mathcal{A}|_{Op(\mathcal{B})}$  which can be lifted to  $f^{-1} : T(\text{Type}(\mathcal{B}), B) \rightarrow T(\text{Type}(\mathcal{B}), A)$ . Then we show that  $f^{-1}$  has the desired properties. Define  $\hat{f}^{-1} : \mathcal{B} \rightarrow \mathcal{A}$  such that  $\hat{f}^{-1}(b) = \{a \mid f(a) = b\}$ . Because  $f$  is surjective (cf. definition 3.2),  $\hat{f}^{-1} \neq \emptyset$  for all  $b \in B$ . Let  $f^{-1}(b) \in \hat{f}^{-1}(b)$  (arbitrary but fixed). Then  $f(f^{-1}(b)) = b$  holds independently of the choice of  $f^{-1}(b) \in \hat{f}^{-1}(b)$  (follows directly from the definition of  $\hat{f}^{-1}$ ).

Let  $t \in T(\text{Type}(\mathcal{B}), B)$ . We show that  $f(\text{eval}_{\mathcal{A}}(f^{-1}(t))) = \text{eval}_{\mathcal{B}}(t)$  for all  $t \in T(\text{Type}(\mathcal{B}), B)$  by induction on the term structure of  $t$ .

**Base Case:**  $t = b$  for some  $b \in B$ .  $f(\text{eval}_{\mathcal{A}}(f^{-1}(b))) = f(\text{eval}_{\mathcal{A}}(a')) = f(a') = f(f^{-1}(b)) = b$  (with  $a' = f^{-1}(b)$ ).

**Induction Case:**  $f(\text{eval}_{\mathcal{A}}(f^{-1}(h(t_1, \dots, t_n)))) =$   
 $= f(\text{eval}_{\mathcal{A}}(h(f^{-1}(t_1), \dots, f^{-1}(t_n))))$  (lift  $f^{-1}$  to terms)  
 $= f(h(\text{eval}_{\mathcal{A}}(f^{-1}(t_1)), \dots, \text{eval}_{\mathcal{A}}(f^{-1}(t_n))))$  (definition of  $\text{eval}_{\mathcal{A}}$ )  
 $= h(f(\text{eval}_{\mathcal{A}}(f^{-1}(t_1))), \dots, f(\text{eval}_{\mathcal{A}}(f^{-1}(t_n))))$  ( $f$  is a homomorphism)  
 $= h(\text{eval}_{\mathcal{B}}(t_1), \dots, \text{eval}_{\mathcal{B}}(t_n))$  (induction assumption)  
 $= \text{eval}_{\mathcal{B}}(h(t_1, \dots, t_n))$  (definition of  $\text{eval}_{\mathcal{B}}$ )  $\square$

Theorem 3.3 states directly that whenever  $\mathcal{A} \succeq \mathcal{B}$ , we can evaluate constant expressions in  $\mathcal{A}$  instead of in  $\mathcal{B}$  because we have transfer functions between  $A$  and  $B$ . Definition 3.2 and theorem 3.3 capture the general case of substitutability between arithmetics in programming languages and target processors. They model not only the mathematical arithmetics  $\mathbb{Z}$  and  $\mathbb{R}$  but also the standard modulo-integer arithmetics as well as saturating arithmetics and floating-point arithmetics. Concerning constant folding, also rare cases are caught: Consider e.g. a compiler which evaluates constant integer expressions with a floating-point arithmetic. If we can state the transfer functions  $f$  and  $f^{-1}$ , we can use the floating-point arithmetic to evaluate the integer expressions. In practice, one often faces the situation that a constant integer expression is to be evaluated within one target modulo-arithmetic but the compiler has only a different modulo-arithmetic. Theorem 3.3 defines the proof obligations to be verified. Sometimes it is not important to ensure substitutability between two arithmetics for arbitrary expressions but only for one specific constant term. This case is described in the following definition:

**Definition 3.4** Let  $\mathcal{A} = (A, F_{\mathcal{A}})$  and  $\mathcal{B} = (B, F_{\mathcal{B}})$  be algebras such that  $\text{Type}(\mathcal{B}) \subseteq \text{Type}(\mathcal{A})$ . Let  $t \in T(\text{Type}(\mathcal{B}), B)$ .  $\mathcal{A}$  is more precise than  $\mathcal{B}$  with respect to  $t$ , denoted by  $\mathcal{A} \succeq_t \mathcal{B}$ , if there exists transfer functions  $f : \mathcal{A} \rightarrow \mathcal{B}$  and  $f^{-1} : \mathcal{B} \rightarrow \mathcal{A}$  such that  $f(\text{eval}_{\mathcal{A}}(f^{-1}(t))) = \text{eval}_{\mathcal{B}}(t)$ .  $\diamond$

If algebra  $\mathcal{A}$  is more precise than algebra  $\mathcal{B}$  wrt. a constant term  $t$ , then we can evaluate  $t$  in  $\mathcal{A}$  instead of in  $\mathcal{B}$  with the transfer functions  $f$  and  $f^{-1}$ .

## 4 Lattices of Integer Arithmetics

In this section, we concentrate on residue class arithmetics, i.e. modulo-arithmetics in  $\mathbb{Z}_n$ ,  $n \in \mathbb{N}^+$ . These arithmetics are contained in programming languages typically in two variants, as signed and unsigned integers. Unsigned integers correspond directly with the set  $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$ . The arithmetical operations on unsigned integers are defined exactly as on  $\mathbb{Z}_n$ . Signed integers represent numbers within the range of  $\{-n/2, \dots, -1, 0, 1, \dots, (n/2)-1\}$ , assuming that  $n$  is even, otherwise in the range  $\{-(n-1)/2, \dots, -1, 0, 1, \dots, (n-1)/2\}$ . If binarily coded in the two's complement representation, numbers starting with a '1' represent negative integers, those starting with '0' represent nonnegative integers.

We can treat both variants, signed and unsigned arithmetics, in a uniform way. Therefore we regard the numbers  $\{0, 1, \dots, n-1\}$  as standard representatives of the congruence classes  $n\mathbb{Z} + r$  of  $\mathbb{Z}$ ,  $0 \leq r \leq n-1$ . Furthermore, we observe that the numbers  $\{-n/2, \dots, -1, 0, 1, \dots, (n/2)-1\}$ <sup>3</sup> denote the same congruence classes whereby  $n\mathbb{Z} + r$  for  $r \in \{(n/2), \dots, n-1\}$  is not represented by its standard representative  $r$  but by the representative  $r - n$ , cf. also figure 1. Each modulo-arithmetic  $\mathbb{Z}_n$ ,  $n \in \mathbb{N}^+$ , is an arithmetic:

**Theorem 4.1** *Each modulo-arithmetic  $\mathbb{Z}_n$  of  $\mathbb{Z}$ ,  $n \in \mathbb{N}^+$ , is an arithmetic of type  $\mathcal{F} = \{+, -, \cdot, 0\}$ .  $\diamond$*

**Proof.** Follows directly from definition 3.1.  $\square$

**Theorem 4.2 (Substitutability of Residue Class Arithmetics)** *Let  $m$ ,  $n \in \mathbb{N}^+$ . If  $n$  divides  $m$ , then  $\mathbb{Z}_m$  is more precise than  $\mathbb{Z}_n$ ,  $\mathbb{Z}_m \succeq \mathbb{Z}_n$ .  $\diamond$*

**Proof.** We need to show that the two requirements of definition 3.2 are fulfilled. The first,  $\text{Type}(\mathbb{Z}_n) \subseteq \text{Type}(\mathbb{Z}_m)$ , holds trivially.

To verify the second requirement, we define a function  $f : \mathbb{Z}_m \longrightarrow \mathbb{Z}_n$  and show that it is a surjective homomorphism. Let  $m = p \cdot n$  ( $p$  exists because  $n$  divides  $m$ ). Each  $x \in \{0, \dots, m-1\}$  can be expressed as  $x = r + n \cdot l$  for some  $l \in \{0, \dots, p-1\}$  and  $0 \leq r \leq n-1$ . Define  $f(x) = r$ . Clearly,  $f$  is surjective because for  $x \in \{0, \dots, n-1\}$ ,  $f(x) = x$  holds trivially. Hence, each element in  $\mathbb{Z}_n$  is the image of at least one element in  $\mathbb{Z}_m$ . To verify that  $f$  is a homomorphism, we need to prove the following four equations:

1.  $f(0_{\mathbb{Z}_m}) = 0_{\mathbb{Z}_n}$
2.  $f(x +_{\mathbb{Z}_m} y) = f(x) +_{\mathbb{Z}_n} f(y)$
3.  $f(-_{\mathbb{Z}_m} x) = -_{\mathbb{Z}_n} f(x)$
4.  $f(x \cdot_{\mathbb{Z}_m} y) = f(x) \cdot_{\mathbb{Z}_n} f(y)$

The first equation holds trivially, the remaining equations are proven below. Thereby we assume that  $x = r + n \cdot l$  and  $y = r' + n \cdot l'$  with  $0 \leq r, r' \leq n-1$  and  $l, l' \in \{0, \dots, p-1\}$ .

**2. Proof of  $f(x +_{\mathbb{Z}_m} y) = f(x) +_{\mathbb{Z}_n} f(y)$ :**  
 $f(x +_{\mathbb{Z}_m} y) = f((r + n \cdot l) +_{\mathbb{Z}_m} (r' + n \cdot l')) =$

<sup>3</sup> For simplicity of notation, we only show the case for  $n$  being even.

$$\begin{aligned}
 &= f((r +_{\mathbb{Z}_m} l \cdot_{\mathbb{Z}_m} n) +_{\mathbb{Z}_m} (r' +_{\mathbb{Z}_m} l' \cdot_{\mathbb{Z}_m} n)) && \text{(because } 0 \leq x, y < m) \\
 &= f(r +_{\mathbb{Z}_m} r' +_{\mathbb{Z}_m} l \cdot_{\mathbb{Z}_m} n +_{\mathbb{Z}_m} l' \cdot_{\mathbb{Z}_m} n) \\
 &= \left\{ \begin{array}{l} r +_{\mathbb{Z}_m} r' \quad \text{if } r +_{\mathbb{Z}_m} r' \leq n - 1 \\ r +_{\mathbb{Z}_m} r' -_{\mathbb{Z}_m} n \text{ otherwise} \end{array} \right\} = r +_{\mathbb{Z}_n} r' = f(x) +_{\mathbb{Z}_n} f(y).
 \end{aligned}$$

**3. Proof of  $f(-_{\mathbb{Z}_m} x) = -_{\mathbb{Z}_n} f(x)$ :**  $f(-_{\mathbb{Z}_m} x) = f(m -_{\mathbb{Z}} x) = f(m -_{\mathbb{Z}} (r +_{\mathbb{Z}} l \cdot_{\mathbb{Z}} n)) = f(m -_{\mathbb{Z}} (l \cdot_{\mathbb{Z}} n) -_{\mathbb{Z}} r) = f(m -_{\mathbb{Z}} r) = f(p \cdot_{\mathbb{Z}} n -_{\mathbb{Z}} r) = f((p -_{\mathbb{Z}} 1) \cdot_{\mathbb{Z}} n +_{\mathbb{Z}} n -_{\mathbb{Z}} r) = n -_{\mathbb{Z}} r = -_{\mathbb{Z}_n} r = -_{\mathbb{Z}_n} f(x).$

**4. Proof of  $f(x \cdot_{\mathbb{Z}_m} y) = f(x) \cdot_{\mathbb{Z}_n} f(y)$ :**

On one hand, we have  $f(x) \cdot_{\mathbb{Z}_n} f(y) = r \cdot_{\mathbb{Z}_n} r' = (r \cdot_{\mathbb{Z}} r') \bmod n$ . On the other hand, we have  $f(x \cdot_{\mathbb{Z}_m} y) = f((r +_{\mathbb{Z}} l \cdot_{\mathbb{Z}} n) \cdot_{\mathbb{Z}_m} (r' +_{\mathbb{Z}} l' \cdot_{\mathbb{Z}} n)) = f(r \cdot_{\mathbb{Z}_m} r')$ .  $r \cdot_{\mathbb{Z}_m} r' = r''$  with  $r \cdot_{\mathbb{Z}} r' = l'' \cdot_{\mathbb{Z}} m +_{\mathbb{Z}} r''$  and  $0 \leq r'' \leq m - 1$ , and  $(r \cdot_{\mathbb{Z}} r') \bmod n = r'''$  with  $r \cdot_{\mathbb{Z}} r' = l''' \cdot_{\mathbb{Z}} n +_{\mathbb{Z}} r'''$  and  $0 \leq r''' \leq n - 1$ . Because  $n$  divides  $m$ , there exists  $q$  such that  $q \cdot_{\mathbb{Z}} l'' = l'''$  and  $r'' \bmod n = r'''$  which completes the proof.  $\square$

**Theorem 4.3**  $\mathbb{Z}$  is more precise than each  $\mathbb{Z}_n$  with  $n \in \mathbb{N}^+$ ,  $\mathbb{Z} \succeq \mathbb{Z}_n$ .

**Proof.** Follows directly from the facts about residue classes in  $\mathbb{Z}$  stated in examples 2.8 and 2.12.  $\square$

**Theorem 4.4 (Lattice of Integer Arithmetics)** The residue class arithmetics  $\mathbb{Z}_n$ ,  $n \in \mathbb{N}^+$ , and  $\mathbb{Z}$  with the partial ordering  $\succeq$  form a lattice  $Int\_Arith = (\{\mathbb{Z}\} \cup \{\mathbb{Z}_n \mid n \in \mathbb{N}^+\}, \succeq)$ .  $\diamond$

**Proof.** We define  $\inf(\mathbb{Z}_n, \mathbb{Z}) = \mathbb{Z}_n$  and  $\sup(\mathbb{Z}_n, \mathbb{Z}) = \mathbb{Z}$  according to the statement of theorem 4.3 above. Moreover, we need to show that for any two arithmetics  $\mathbb{Z}_n$  and  $\mathbb{Z}_m$ ,  $\inf(\mathbb{Z}_n, \mathbb{Z}_m)$  and  $\sup(\mathbb{Z}_n, \mathbb{Z}_m)$  exist. Therefore we prove the following whereby “gcd” stands for greatest common divisor and “lcm” for least common multiple.

$$\inf(\mathbb{Z}_n, \mathbb{Z}_m) = \mathbb{Z}_{\gcd(n, m)} \quad \text{and} \quad \sup(\mathbb{Z}_n, \mathbb{Z}_m) = \mathbb{Z}_{\text{lcm}(n, m)}$$

We only prove the case for  $\inf(\mathbb{Z}_n, \mathbb{Z}_m)$  because the case for the supremum is analogous. Clearly,  $\mathbb{Z}_n \succeq \mathbb{Z}_{\gcd(n, m)}$  and  $\mathbb{Z}_m \succeq \mathbb{Z}_{\gcd(n, m)}$ . Moreover, there exists no other  $\mathbb{Z}_k$  with  $k > \gcd(m, n)$  such that  $\mathbb{Z}_n \succeq \mathbb{Z}_k$  and  $\mathbb{Z}_m \succeq \mathbb{Z}_k$  because then  $k$  cannot divide  $n$  and  $m$ .  $\square$

The top element of the lattice  $Int\_Arith$  is  $\mathbb{Z}$ , the bottom element is  $\mathbb{Z}_1$ . The lattice  $Int\_Arith$  is a complete lattice because for every subset  $A$  of  $\{\mathbb{Z}\} \cup \{\mathbb{Z}_n \mid n \in \mathbb{N}^+\}$ ,  $\text{Inf}(A)$  and  $\text{Sup}(A)$  exist. The next theorem states the connection of  $Int\_Arith$  with the lattice of congruence relations on  $\mathbb{Z}$ , **Con**  $\mathbb{Z}$ . This theorem follows from the fact that the residue classes  $n\mathbb{Z}$  for  $n \in \mathbb{N}^+$  are exactly the ideals on  $\mathbb{Z}$ , as explained in example 2.12. Each ideal defines a congruence relation  $\theta_n$  whose congruence classes are  $n\mathbb{Z} + r$ ,  $0 \leq r \leq n - 1$ , cf. figure 1, with  $n\mathbb{Z} + r = \{x \mid x \bmod n = r\}$ . Vice versa, each congruence relation on  $\mathbb{Z}$  defines an ideal which is the congruence class containing 0.

**Theorem 4.5** The lattice  $Int\_Arith$  is isomorphic to the dual of the lattice **Con**  $\mathbb{Z}$  of congruence relations on  $\mathbb{Z}$ .

**Proof.** Let  $\theta_n = \{(x, y) \mid x \bmod n = y \bmod n\}$ . We define the function  $f : \text{Int\_Arith} \longrightarrow \text{Con } \mathbb{Z}$  with  $f(\mathbb{Z}_n) = \theta_n$  and  $f(\mathbb{Z}) = \{(z, z) \mid z \in \mathbb{Z}\} =: \theta_\infty$  (i.e. the congruence relation where only identical integers are equivalent) and the function  $f^{-1} : \text{Con } \mathbb{Z} \longrightarrow \text{Int\_Arith}$  with  $f^{-1}(\theta_n) = \mathbb{Z}_n$  and  $f^{-1}(\theta_\infty) = \mathbb{Z}$ . Clearly,  $f$  and  $f^{-1}$  are surjective functions and  $f^{-1}$  is the inverse function of  $f$ . We need to show that  $f$  is an isomorphism, i.e.,  $\mathbb{Z}_m \succeq \mathbb{Z}_n$  iff  $\theta_m \subseteq \theta_n$ .

“ $\Rightarrow$ ”: **Assume  $\mathbb{Z}_m \succeq \mathbb{Z}_n$  and show  $\theta_m \subseteq \theta_n$ :**

Assume  $\mathbb{Z}_m \succeq \mathbb{Z}_n$ . Then there is a surjective homomorphism  $f : \mathbb{Z}_m \longrightarrow \mathbb{Z}_n$ . Furthermore, the surjective homomorphisms  $g : \mathbb{Z} \longrightarrow \mathbb{Z}_m$  and  $h : \mathbb{Z} \longrightarrow \mathbb{Z}_n$  exist because  $\mathbb{Z} \succeq \mathbb{Z}_n$  and  $\mathbb{Z} \succeq \mathbb{Z}_m$ .  $h = f \circ g$  holds. Moreover,  $\ker(g) = \theta_m$  and  $\ker(h) = \theta_n$ . Assume that  $(x, y) \in \theta_m = \ker(g)$  exists but  $(x, y) \notin \theta_n = \ker(h)$ . This is a contradiction because all elements which are mapped to the same image by  $g$  must also be mapped to the same image by  $f \circ g = h$ . Hence,  $\theta_m \subseteq \theta_n$  must hold.

“ $\Leftarrow$ ”: **Assume  $\theta_m \subseteq \theta_n$  and show  $\mathbb{Z}_m \succeq \mathbb{Z}_n$ :**

Assume that  $\theta_m \subseteq \theta_n$ . To show that  $\mathbb{Z}_m \succeq \mathbb{Z}_n$  holds we prove that  $n$  divides  $m$ . To prove this, we assume the contrary,  $n$  does not divide  $m$ , and show that this results in a contradiction:  $(m, 0) \in \theta_m$  but  $(m, 0) \notin \theta_n$  because  $n$  does not divide  $m$ . But this is a contradiction to the assumption that  $\theta_m \subseteq \theta_n$ . Hence, we conclude that  $n$  divides  $m$  and that  $\mathbb{Z}_m \succeq \mathbb{Z}_n$ .  $\square$

From theorem 4.5, it follows directly that the condition stated in theorem 4.2 is not only a sufficient but also a necessary criterion:

**Corollary 4.6** *Let  $m, n \in \mathbb{N}^+$ .  $n$  divides  $m$  iff  $\mathbb{Z}_m \succeq \mathbb{Z}_n$ .*  $\diamond$

One might wonder why we did not include the integer operations “mod” and “div” which are available in many programming languages into the type of modulo arithmetics when stating theorem 4.1. The following counterexample shows that under this assumption, we would not be able to prove a criterion for substitutability similar to that in corollary 4.6:  $(4 +_{\mathbb{Z}_8} 5) \bmod_{\mathbb{Z}_8} 7 = 1$  but  $(4 +_{\mathbb{Z}_{16}} 5) \bmod_{\mathbb{Z}_{16}} 7 = 2$ , and  $(4 +_{\mathbb{Z}_8} 5) \text{div}_{\mathbb{Z}_8} 2 = 0$  but  $(4 +_{\mathbb{Z}_{16}} 5) \text{div}_{\mathbb{Z}_{16}} 2 = 4$ .

Corollary 4.6 gives us an efficiently decidable criterion for the substitutability of an integer arithmetic in a target processor by the integer arithmetic in a compiler. Constant expressions of the integer arithmetic  $\mathbb{Z}_n$  can be evaluated in the integer arithmetic  $\mathbb{Z}_m$  iff  $n$  divides  $m$ . In modern processor architectures,  $n$  and  $m$  are always a power of 2. For them, this criterion states that constant expressions can be evaluated if the representation of numbers in the compiler arithmetic is equal or larger than the representation of numbers in the target processor and if both compute values according to the modulo-arithmetic (which is not as clear as it may seem, cf. our discussion in the next section). In the following section, we discuss the C and Java integer arithmetics and show how this criterion can be applied to classify valid constant folding optimizations in their compilers. These considerations also reveal that the C language standard [ISO99] contains some dangerous definitions concerning integer arithmetic.

Java Integers		C Integers	
Signed	Unsigned	Signed	Unsigned
long (64-bit)		long long int	unsigned long long int
int (32-bit)		long int	unsigned long int
short (16-bit)	char (16-bit)	int	unsigned int
byte (8-bit)		short int	unsigned short int
		signed char	unsigned char

Fig. 2. Java and C Integer Data Types

## 5 Integer Arithmetics in Java and C

The Java language specification defines exactly how integer numbers are represented and how integer arithmetic expressions are to be evaluated. This is an important property of Java as this programming language has been designed to be used in distributed applications on the Internet. A Java program is required to produce the same result independently of the target machine executing it. In contrast, C (and the majority of widely-used imperative and object-oriented programming languages) is more sloppy and leaves many important characteristics open. The intention behind this inaccurate language specification is clear. The same C programs are supposed to run on a 16-bit, 32-bit, or even 64-bit architecture by instantiating the integer arithmetics of the source programs with the arithmetic operations built-in in the target processor. This leads to much more efficient code because it can use the available machine operations directly. As long as the integer computations deal only with numbers being “sufficiently small”, no inconsistencies will arise. In this sense, the C integer arithmetic is a placeholder which is not defined exactly by the programming language specification but is only completely instantiated by determining the target machine. In this section, we discuss both C and Java integer arithmetics. Thereby we show that the C integer arithmetic bears potential sources of incorrect arithmetic behavior.

Java precisely defines how integers are represented and how integer arithmetic is to be computed. The values of all signed integers are two’s complement representations of the length as listed in figure 2. Char is the only unsigned integer type. Its values represent Unicode characters, from ‘\u0000’ to ‘\uffff’, i.e. from 0 to  $2^{16} - 1$ . If an integer operator has an operand of type long, then the other operand is also converted to type long. Otherwise the operation is performed on operands of type int, if necessary shorter operands are converted into int. The conversion rules are exactly specified, cf. [ESGB00].

The exact specification of Java integer arithmetic determines exactly the values of constant integer expressions which are computed by the compiled

programs, independently of the target machine executing them. They are computed within the arithmetic  $\mathbb{Z}_{64}$  if it is an operation on long integers, otherwise in  $\mathbb{Z}_{32}$ . Recall that two's complement in  $\mathbb{Z}_{2^n}$  is only the choice of non-standard representatives for the congruence classes  $n\mathbb{Z} + r$  for  $r \geq 2^{n-1}$ , cf. our explanation at the beginning of section 4. From corollary 4.6, it follows that we can evaluate constant integer expressions in Java programs already at compile time iff one of the following two conditions holds:

- (i) The expression is to be evaluated in  $\mathbb{Z}_{64}$  and the compiler uses an arithmetic  $\mathbb{Z}_{n \cdot 64}$ ,  $n \in \mathbb{N}^+$ , or
- (ii) the expression is to be evaluated in  $\mathbb{Z}_{32}$  and the compiler uses an arithmetic  $\mathbb{Z}_{n \cdot 32}$ ,  $n \in \mathbb{N}^+$ .

The C language specification [ISO99] does not define integer values and integer arithmetic as exactly as the Java specification. We do not give all details here but discuss only the most important characteristics and their implications on compiler arithmetics. C has two kinds of integer values, signed and unsigned ones, cf. figure 2. The C language specification defines a header file `<limits.h>` which determines the minimum and maximum values representable in the respective integer type. A given compiler is supposed to provide this file such that its specific ranges of integer values contains the ranges determined in the C specification. The C specification requires that for each signed integer type, there is a corresponding but different unsigned integer type of the same size. Values of type integer have the “natural sizes suggested by the architecture of the execution environment (large enough to contain any value in the range `Int_Min` to `Int_Max` as defined in the header `<limits.h>`)” (cf. 6.2.5 of [ISO99]). Unsigned integers represent values in the range of  $0, \dots, 2^N - 1$  where  $N$  is the length of representation. The header file `<limits.h>` determines the minimum range of unsigned integers as  $0, \dots, 2^N - 1$ , where  $N$  is 8 for `char`, 16 for `short` and `int`, 32 for `long int`, and 64 for `long long int`. For signed integers, the ranges  $-(2^{N-1} - 1), \dots, 2^{N-1} - 1$  are specified,  $N$  defined as for unsigned integers. E.g. the GNU C compiler [Pro02] redefines the ranges as  $-2^{N-1}, \dots, 2^{N-1} - 1$  according to  $\mathbb{Z}_{2^N}$ , with  $N = 32$  for `int`.

These definitions in the C specification, especially those in `<limits.h>`, bear two potential sources of unexpected behavior if constant folding is performed. The first concerns the incomplete specification of the target arithmetic. If the target architecture adheres exactly to the ranges in `<limits.h>`, then arithmetic is not computed according to  $\mathbb{Z}_{2^N}$  because the element  $-2^{N-1}$  does not exist. It remains unclear whether the C specification intended to require a  $\mathbb{Z}_{2^N-1}$  or a  $\mathbb{Z}_{2^N}$  arithmetic. Hence, the compiler arithmetic cannot evaluate arbitrary integer expressions at compile time. Only those can be computed for which the requirements stated in definition 3.4 can be verified. For example, if the integers in the constant expression are very small such that the distinction between  $\mathbb{Z}_{2^N-1}$  and  $\mathbb{Z}_{2^N}$  does not matter, then the expression can be evaluated. The second source of incorrectness of constant folding arises

from the fact that the integer ranges can be extended arbitrarily by the target machine. E.g. it would be conform with the C specification to extend the ranges such that much more negative than positive numbers are contained. While this or similar extensions still fit into the residue class setting (we can choose the representatives of the equivalence classes arbitrarily), it might become a practical problem: The binary representation of integers would not have the well-known interpretation any more that numbers starting with a ‘1’ are negative and all others are non-negative. Compiler programmers might not expect non-standard interpretations of the integers. Hence, these allowed unusual interpretations are a potential source of errors.

## 6 Formalization in Isabelle/HOL

We have formalized our main result concerning substitutability of integer arithmetics within the Isabelle/HOL [NPW02] system. Isabelle is a generic theorem prover. It can be instantiated with different logics, whereas Isabelle/HOL, simply typed higher order logic, is the one most widely used. Our main result is stated in theorem 4.2 and says that  $\mathbb{Z}_m$  is more precise than  $\mathbb{Z}_n$  if  $n$  divides  $m$ . If one wants to formally verify a compiler doing integer constant folding, then our Isabelle proof can become part of this overall correctness proof. Our Isabelle proof is generic as it does not instantiate the numbers  $n$  and  $m$ . In this section, we explain the main data structures and proof steps of our Isabelle formalization. An Isabelle proof document contains data type definitions, function and constant definitions and lemmata. A lemma can be verified by applying certain proof techniques as e.g. induction, case distinction, or the application of already verified lemmata.

We have modelled constant expressions as trees:

```
datatype Etree = Leaf int | Node operator Etree Etree
datatype operator = Add | Sub | Mult
```

We have defined functions which evaluate constant expression trees. The function *calc* evaluates trees within the ring  $\mathbb{Z}$  of the integer numbers. *calcm* evaluates in the arithmetic of  $\mathbb{Z}_m$  and *calcmn* in  $\mathbb{Z}_{m \cdot n}$ . We give here only the definitions for *calc* and *calcm*. *calcmn* is analogously defined.

<pre>consts calc :: "Etree <math>\Rightarrow</math> int" primrec "calc (Leaf a) = a" "calc (Node ox a b) = (case ox of Add <math>\Rightarrow</math> (calc a) + (calc b)  Sub <math>\Rightarrow</math> (calc a) - (calc b)  Mult <math>\Rightarrow</math> (calc a) * (calc b))"</pre>	<pre>consts calcm :: "Etree <math>\Rightarrow</math> int <math>\Rightarrow</math> int" primrec "calcm (Leaf a) m = a mod m" "calcm (Node ox a b) m = (case ox of Add <math>\Rightarrow</math> (((calcm a m) + (calcm b m)) mod m)  Sub <math>\Rightarrow</math> (((calcm a m) - (calcm b m)) mod m)  Mult <math>\Rightarrow</math> (((calcm a m) * (calcm b m)) mod m))"</pre>
--	--

Our first lemma states that results of operations in  $\mathbb{Z} \bmod m$  are the same as taking the operands mod  $m$  and then computing the result in  $\mathbb{Z}_m$ .

*lemma "(calc a)mod m = (calcm a (m :: int))"*

The lemma can be verified by induction on  $a$ , "apply (induct\_tac a)". Isabelle creates these proof obligations:

1.  $\bigwedge int. \text{calc } (Leaf\ int) \bmod m = \text{calcm } (Leaf\ int)$
2.  $\bigwedge operator\ Etree1\ Etree2.$   
 $[[\text{calc } Etree1 \bmod m = \text{calcm } Etree1\ m;$   
 $\text{calc } Etree2 \bmod m = \text{calcm } Etree2\ m]]$   
 $\text{calc } (Node\ operator\ Etree1\ Etree2) \bmod m =$   
 $\text{calcm } (Node\ operator\ Etree1\ Etree2)\ m$

The base case of the induction can be verified directly using the definitions of *calc* and *calcm*. Isabelle does this step automatically by using the tactic "apply auto". The remaining proof obligations can be verified by a case distinction over the possible operators. First we pick the *Add*-operator and start a case distinction, "apply (case\_tac "operator = Add)". Isabelle produces this result (we omit the other case "operator  $\neq$  Add" for space reasons):

$\bigwedge Etree1\ Etree2.$   
 $[[\text{calc } Etree1 \bmod m = \text{calcm } Etree1\ m;$   
 $\text{calc } Etree2 \bmod m = \text{calcm } Etree2\ m]]$   
 $\implies (\text{calc } Etree1 + \text{calc } Etree2) \bmod m =$   
 $(\text{calcm } Etree1\ m + \text{calcm } Etree2\ m) \bmod m$

The following lemma (available in Isabelle) is used in the next proof step:

*lemma "(a + b)mod m = (a mod m + b mod m)mod (m :: int)"*

Isabelle can completely prove this remaining proof obligation automatically. The two other cases are slightly more complicated because lemmata as the one used above need to be proved before. We have also verified that *calcm* can be replaced by *calcmn*.

## 7 Related Work

Correctness of compilers has been investigated in many research projects. Nevertheless, as to the authors' knowledge, there has been no research investigating and classifying different arithmetics with respect to their substitutability (except for the general requirement that the translated programs must show the same behavior as the source programs). A very early research considering correctness of compiling arithmetics is [MP67] which verifies the translation of arithmetic expressions into machine code, but without paying attention to the fact that the source and target arithmetic may be different.

The german Verifix project [GZ99] has the goal of constructing correct compilers. This project has achieved progress by establishing the claim that it is possible to build provably correct compilers within the standard framework of compiler construction. In [Nec00], it is shown how some backend optimizations of the GCC can be validated. Proof-carrying code [NL98] is

another weaker approach to the construction of correct compilers which guarantees that the generated code fulfills certain necessary correctness conditions. Pnueli [PSS98,ZPL01] also addresses the problem of constructing correct compilers. [GGB02] investigates verification of compiler optimizations specific for embedded processors. None of these works addresses the problem of dealing with different arithmetics in programming languages and their compilers. In particular, none of these works establishes a general substitutability criterion between different arithmetics.

## 8 Conclusions

In this paper, we have stated a general sufficient criterion for substitutability of arithmetics. Therefore we defined arithmetics as universal algebras of certain types. Our substitutability criterion defines an order relation on the arithmetics. Concerning integer arithmetics, we have shown that the residue class arithmetics form a lattice which is isomorphic to the dual lattice of the congruence relations of  $\mathbb{Z}$ . This characterization has given us a sufficient and necessary criterion for substitutability. We discussed and compared the integer arithmetics of Java and C. Their characteristics display the different intentions and purposes for using Java or C as implementation language. Java is designed for distributed applications on the internet. Java program behavior must be uniquely determined independently of the executing machine. In contrast, C is often used for system implementations close to the machine code level. These C programs must be as efficient as possible and therefore use the available machine arithmetic. Hence, in C, the arithmetic operations are not fully specified and serve as a placeholder for various machine arithmetics. We have argued that this incomplete specification of C integer arithmetic is a potential source of incorrect compiler behavior. Nevertheless, our results state a simple criterion for substitutability and may help in reducing compiler mistakes. We have formalized our criterion for substitutability of integer arithmetics in the Isabelle/HOL system, an interactive higher-order theorem prover. This proof is generic as it does not specify the absolute sizes of the involved integer arithmetics. This formal proof may become part of a formal proof of correctness of a compiler performing constant folding.

In future work, we want to investigate further arithmetics as e.g. saturating integer arithmetic. Saturating arithmetic is not a ring any more but still an arithmetic in the sense of our formalization. We also want to consider floating-point arithmetics. It is an open question if there are different floating-point arithmetics being in the substitutability relation. Most probably we need to widen our definition of substitutability and parameterize it with the relative size of rounding errors to classify floating-point arithmetics adequately.

## References

- [BS00] Stanley Burris and H.P. Sankappanavar. A Course in Universal Algebra, 2000. Millenium Edition. Originally published by Springer in 1981.
- [ESGB00] Bill Joy (Editor), Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.
- [GGB02] Sabine Glesner, Rubino Geiß, and Boris Boesler. Verified Code Generation for Embedded Systems. In *Proc. COCV-Workshop, 2002*. Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 65.2.
- [GZ99] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In *Correct System Design*. Springer-Verlag, LNCS 1710, 1999.
- [Ihr88] Thomas Ihringer. *Allgemeine Algebra*. Teubner, 1988.
- [ISO99] ISO/IEC. Int'l Standard ISO/IEC 9899:1999, Programming languages – C. Ref. no. ISO/IEC 9899:1999(E), 1999. 2nd edition 1999-12-01.
- [Lan79] S. Lang. *Algebraische Strukturen*. Vandenhoeck und Ruprecht, 1979.
- [MP67] John McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science, Proc. Symp. in Appl. Math.*, Am. Math. Soc., 1967.
- [Nec00] George C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings PLDI'00*, 2000.
- [NL98] George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. In *Proc. PLDI'98*, 1998.
- [NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, LNCS 2283, 2002.
- [Poo94] Martin Poole. Problems with compiler optimisation (Pentium related). Forum on Risks to the Public in Computers and Related Systems, Volume 16: Issue 63, 1994. available at <http://catless.ncl.ac.uk/Risks/16.63.html#subj8>.
- [Pro02] GNU Project. GCC Home Page. <http://gcc.gnu.org/>, 2002. Version gcc version 2.96 20000731 (Red Hat Linux 7.1 2.96-98).
- [PSS98] A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool (cvt.). *Int'l Journal Software Tools for Technology Transfer*, 2(2):192–201, 1998.
- [ZPL01] L. Zuck, A. Pnueli, and R. Leviathan. Validation of Optimizing Compilers. Technical Report MCS01-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, August 2001.

## A Isabelle Proof

In this appendix, we list our complete Isabelle proof skript. It is not necessary to read it in order to understand the paper. This appendix is only a complementary information.

*Etree.thy*

theory Etree = Main:

**(\* Basic stuff\*)**

lemma l1: "(a + b) mod m = (a mod m + b mod m) mod (m::int)"

apply (rule zmod\_zadd1\_eq)

done

lemma l2a: "(a::int) - b = a + -b"

apply (auto)

done

lemma l2b: "m - b mod (m::int) = m + (- (b mod m))"

apply (auto)

done

lemma l2c: "(m + - (b mod m)) mod m = (m mod m + - (b mod m) mod m) mod (m::int)"

apply (rule zmod\_zadd1\_eq)

done

lemma l2d: "(a mod m - b mod m) mod m = (a mod m + (- (b mod m))) mod (m::int) "

apply (auto)

done

lemma l2e: "(a mod m + (- (b mod m)) mod m) mod (m::int) = (a mod m + (- (b mod m))) mod (m::int) "

apply (subst zmod\_zadd1\_eq)

apply (auto)

done

lemma l2f: "((a::int) mod m - b mod m) mod m = (a - b) mod m"

apply (subst l2a)

apply (subst zmod\_zadd1\_eq)

apply (subst zmod\_zminus1\_eq\_if)

apply (case\_tac "b mod m = 0")

apply (auto)

```

apply (subst zmod_zadd1_eq)
apply (subst l2b)
apply (subst zmod_zadd1_eq)
apply (subst l2c)
apply (auto)
apply (subst l2e)
apply (auto)
done

```

```

lemma l2: “(a - b) mod m = ((a::int) mod m - b mod m) mod m “
apply (subst l2f)
apply (auto)
done

```

```

lemma l3a:”a mod m * (b mod m) mod (m::int) = a * b mod m”
apply (subst zmod_zmult1_eq)
apply (subst zmult_commute)
apply (subst zmod_zmult1_eq)
apply (subst zmult_commute)
apply (auto)
done

```

```

lemma l3:”(a * b) mod m = ((a mod m) * (b mod m)) mod (m::int)”
apply(subst l3a)
apply (auto)
done

```

```

lemma l4a: “(m * (a div m mod n) + a mod m) mod m = (m * (a div m mod n)
mod m + a mod m mod m) mod (m::int)”
apply (simp add: zmod_zadd1_eq)
done

```

```

lemma l4b: “m * (a div m mod n) mod m = (0::int)”
apply (auto)
done

```

```

lemma l4: “(1 ≤ n) ⇒ (a::int) mod m = a mod (m * n) mod m “
apply (subst zmod_zmult2_eq)
apply (auto)
apply (subst l4a)
apply (subst l4b)
apply (auto)
done

```

**(\*Our Expression Tree\*)**

```
datatype operator = Add | Sub | Mult
datatype Etree = Leaf int | Node operator Etree Etree
```

**(\* Ordinary Integer Arithmetics\*)**

```
consts
calc :: "Etree  $\Rightarrow$  int"
primrec
"calc (Leaf a) = a"
"calc (Node ox a b) = (case ox of
Add  $\Rightarrow$  (calc a) + (calc b) |
Sub  $\Rightarrow$  (calc a) - (calc b) |
Mult  $\Rightarrow$  (calc a) * (calc b) )"
```

**(\* Ordinary Integer Arithmetics mod m\*)**

```
consts
calcm :: "Etree  $\Rightarrow$  int  $\Rightarrow$  int"
primrec
"calcm (Leaf a) m = a mod m"
"calcm (Node ox a b) m = (case ox of
Add  $\Rightarrow$  (((calcm a m) + (calcm b m)) mod m) |
Sub  $\Rightarrow$  (((calcm a m) - (calcm b m)) mod m) |
Mult  $\Rightarrow$  (((calcm a m) * (calcm b m)) mod m) )"
```

lemma m1: "(calc a + calc b) mod m = (calc a mod m + calc b mod m) mod m"

```
apply (rule zmod_zadd1_eq)
done
```

lemma m1p: "(calc Etree1 - calc Etree2) mod m = (calc Etree1 mod m - calc Etree2 mod m) mod m"

```
apply (rule l2)
done
```

lemma m1pp: "calc Etree1 \* calc Etree2 mod m = (calc Etree1 mod m) \* (calc Etree2 mod m) mod m"

```
apply (rule l3)
done
```

**(\* You can either calculate an Expression Tree using normal integer arithmetics taking the result mod m, but you can also use mod m - integer arithmetics, giving the same result \*)**

lemma "(calc a) mod m = (calcm a (m::int))"

```

apply (induct_tac a)
apply (auto)
apply (case_tac "operator = Add")
apply (auto)
apply (subst m1)
apply (auto)
apply (case_tac "operator = Sub")
apply (auto)
apply (subst m1p)
apply (auto)
apply (case_tac "operator = Mult")
apply (auto)
apply (subst m1pp)
apply (auto)
apply (case_tac operator)
apply (auto)
done

```

consts

calcmn :: "Etree  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  int"

primrec

"calcmn (Leaf a) m n = a mod (m\*n)"

"calcmn (Node ox a b) m n = ( case ox of

Add  $\Rightarrow$  ((calcmn a m n) + (calcmn b m n)) mod (m\*n) |

Sub  $\Rightarrow$  ((calcmn a m n) - (calcmn b m n)) mod (m\*n) |

Mult  $\Rightarrow$  ((calcmn a m n) \* (calcmn b m n)) mod (m\*n)

)

"

lemma m2: "(1  $\leq$  n)  $\implies$  (calcmn Etree1 m n + calcmn Etree2 m n) mod (m \* n) mod m = (calcmn Etree1 m n + calcmn Etree2 m n) mod m"

apply (subst l4)

apply (auto)

done

lemma m3: "(calcmn Etree1 m n + calcmn Etree2 m n) mod m = (calcmn Etree1 m n mod m + calcmn Etree2 m n mod m) mod m"

apply (rule zmod\_zadd1\_eq)

done

lemma m2p: "(1  $\leq$  n)  $\implies$  (calcmn Etree1 m n - calcmn Etree2 m n) mod (m \* n) mod m = (calcmn Etree1 m n - calcmn Etree2 m n) mod m"

apply (subst l4)

apply (auto)

done

lemma m3p: “(calcmn Vtree1 m n - calcmn Vtree2 m n) mod m = (calcmn Vtree1 m n mod m - calcmn Vtree2 m n mod m) mod m”

apply (rule l2)

done

lemma m2pp: “(1 ≤ n) ⇒ (calcmn Etree1 m n \* calcmn Etree2 m n) mod (m \* n) mod m = (calcmn Etree1 m n \* calcmn Etree2 m n) mod m”

apply (subst l4)

apply (auto)

done

lemma m3ppa: “ ((calcmn Etree1 m n mod m) \* (calcmn Etree2 m n mod m)) mod m = (calcmn Etree1 m n \* calcmn Etree2 m n) mod m “

apply (subst l3)

apply (auto)

done

lemma m3pp: “(calcmn Etree1 m n \* calcmn Etree2 m n) mod m = ((calcmn Etree1 m n mod m) \* (calcmn Etree2 m n mod m)) mod m”

apply (subst m3ppa)

apply (auto)

done

**(\* You can either calculate an Expression Tree using integer arithmetics mod m\*n taking the result mod m, but you can also use mod m - integer arithmetics, giving the same result \*)**

lemma “(1 ≤ n) ⇒ (calcmn a m n) mod m = (calcm a (m::int))”

apply (induct\_tac a)

apply (auto)

apply (subst l4)

apply (auto)

apply (case\_tac “operator = Add”)

apply (auto)

apply (subst m2)

apply (auto)

apply (subst m3)

apply (auto)

apply (case\_tac “operator = Sub”)

apply (auto)

apply (subst m2p)

apply (auto)

```
apply (subst m3p)
apply (auto)
apply (case_tac "operator = Mult")
apply (auto)
apply (subst m2pp)
apply (auto)
apply (subst m3pp)
apply (auto)
apply (case_tac operator)
apply (auto)

done
end
```