Formal Numerical Program Analysis[*]

William M. Farmer and F. Javier Thayer

The MITRE Corporation
202 Burlington Road, A118
Bedford, MA 01730-1420

{farmer,jt}@mitre.org

12 October 1994

Abstract

This paper describes a method for formally analyzing numerical programs and a software system that implements the method. The software system translates a purely functional Pre-Scheme program that manipulates machine integers into a representation in the IMPS Interactive Mathematical Proof System. The correctness of the Pre-Scheme program is analyzed by stating and proving conjectures about the IMPS representation using the IMPS theorem proving facility.

---

[*] This work was carried out in FY94 under MITRE's System Security Analysis project (project number: 8360; sponsor: NSA/C62).

Contents

1 Introduction **1**

2 Pre-Scheme **2**

3 IMPS **3**

4 The IMPS Theory of Machine Arithmetic **4**

5 The Pre-Scheme-to-IMPS Translator **5**

6 Examples **7**
- 6.1 Even and Odd Testers 8
  - 6.1.1 The Pre-Scheme Program 8
  - 6.1.2 Discussion 8
- 6.2 Recursive Factorial Function 9
  - 6.2.1 The Pre-Scheme Program 9
  - 6.2.2 Discussion 9
- 6.3 Iterative Factorial Function 9
  - 6.3.1 The Pre-Scheme Program 9
  - 6.3.2 Discussion 9
- 6.4 Fibonacci Function 10
  - 6.4.1 The Pre-Scheme Program 10
  - 6.4.2 Discussion 10
- 6.5 Greatest Common Denominator 10
  - 6.5.1 The Pre-Scheme Program 10
  - 6.5.2 Discussion 11

7 Conclusion **11**

A The File for Machine-Arithmetic **13**

B The File for Even and Odd Testers **23**

C The File for Recursive Factorial Function **33**

D The File for Iterative Factorial Function **36**

E The File for Fibonacci Function **39**

F The File for Greatest Common Denominator **48**

References **53**

1 Introduction

This paper describes a method for formally analyzing numerical programs and a software system that implements the method. By a numerical program we mean one which uses numerical datatypes such as machine integers or floating-point reals. A formal analysis of a program means the investigation of a representation of the program as some mathematical object, such as a lambda expression, in a formal mathematical theory. The representation is created in a formal theory so that we can state and prove properties of the program using conventional mathematical techniques with the assistance of a mechanized mathematics system. In particular, we can attempt to produce a machine-checked proof that the program computes an abstractly specified mathematical function, or is within certain bounds of that function.

An enormous number of commonly used, practical programs perform computations involving numerical datatypes—for example, programs for computing solutions of all sorts of equations (algebraic equations, differential equations), for computing transforms (Fourier, Laplace, Mellin), and for real-time control. Though much is known about such programs in a practical sense, a precise mathematical analysis of even a simple numerical program is very difficult for the following reasons:

- Commonly used numerical datatypes are *approximations* of familiar mathematical objects such as the ring of integers or the field of real numbers.
- The operations on these numerical datatypes may have overflow (or underflow). Moreover, common algebraic laws, such as the associative law, may not be valid.
- The programs themselves may compute mathematical objects or approximations thereof requiring a large amount of mathematical machinery to specify.
- The mathematical formalization of how the computed solutions of a problem approximate the abstractly defined solution is usually itself more difficult than the theory of the exact solution.

The basic idea of our method is to write a numerical program in the Pre-Scheme programming language [9] and then translate it into a representation in the IMPS Interactive Mathematical Proof System [2] so that conjectures concerning the correctness of the program can be investigated with the help of IMPS. The representation of the Pre-Scheme program in IMPS is based on the standard for numerical datatypes proposed by M. Payne, C. Schaffert, and B. Wichmann [8]. We have produced a software system that implements the method for purely functional Pre-Scheme programs that manipulate just machine integers. Restricting our attention to machine integers allowed us to demonstrate our method while avoiding the complexity of more sophisticated numerical datatypes such as the floating-point reals. In subsequent work, we would like to extend our system to handle purely functional Pre-Scheme programs that manipulate both machine integers and floating-point reals.

The paper is organized as follows. Some background on the Pre-Scheme programming language and the IMPS system is given in section 2 and section 3, respectively. The IMPS theory of machine arithmetic, which is used as the basis for representing numerical Pre-Scheme programs, is discussed in section 4; the complete specification of the theory is presented in appendix A. Section 5 describes the software that translates a numerical Pre-Scheme program into an IMPS representation. Several examples of how the method is employed are given in section 6; the details of the examples are presented in appendices B, C, D, E, and F. Section 7 contains some final remarks.

2 Pre-Scheme

Pre-Scheme is a programming language invented by R. Kelsey [6] and J. Rees which is intended for systems programming. Its syntax is essentially the same as the syntax of Scheme so that Pre-Scheme programs can be run and debugged as if they were ordinary Scheme programs. Its semantics is also very similar to the semantics of Scheme; the semantics of both Scheme and Pre-Scheme can be defined succinctly by means of denotational semantics [5, 9]. Pre-Scheme can be executed using only a C-like run-time system, in which, for example, there is no run-time type checking and no garbage collection.

Our software system uses a version of Pre-Scheme called VLISP Pre-Scheme that was developed and implemented under MITRE's VLISP project [4]. The compiler for VLISP Pre-Scheme, which is written in Scheme, was verified under VLISP [7]. VLISP Pre-Scheme has over 50 standard primitive operators, including the following operators for machine arithmetic: `=`, `<`, `<=`, `>`, `>=`, `+`, `*`, `-`, `abs`, `quotient`, and `remainder`.[1]

---

[1] The primitive operator `-` is overloaded: it represents subtraction when it has two

Compilation is performed in two main stages. The first stage translates a VLISP Pre-Scheme program into a language called Simple Pre-Scheme by expanding derived syntax, changing bound variables, and applying a series of meaning-refining transformations.[2] Simple Pre-Scheme is a sublanguage of VLISP Pre-Scheme with the following two properties:

(1) Each Simple Pre-Scheme program has a very restricted syntactic form. In particular, a Simple Pre-Scheme program has exactly one occurrence of `letrec` which is at the top level, and `lambda` expressions may occur only as initializers in `letrec` bindings or in the operator position of a procedure call. (The `lambda` construct in Pre-Scheme is used to create a procedure, while the `letrec` construct is used to define a list of local procedures that may be mutually recursive.)

(2) Each Simple Pre-Scheme program is strongly typed.

These two properties make it easy to compile Simple Pre-Scheme programs into machine code. This first step of the compiler will not succeed on all VLISP Pre-Scheme programs, and hence, the compiler does not accept the entire VLISP Pre-Scheme language.

The second stage translates a Simple Pre-Scheme program either into C or into assembly language for a MIPS computer architecture.

3 IMPS

IMPS is a mathematical reasoning environment that is intended to support traditional mathematical techniques and styles of practice. The system consists of a database of mathematics (represented as a network of axiomatic theories linked by theory interpretations) and a collection of tools for exploring, applying, extending, and communicating the mathematics in the database. One of the chief tools is a facility for interactively developing formal proofs. The IMPS theory library currently contains significant portions of logic, algebra, and analysis with over 1100 replayable proofs. The IMPS logic is a version of simple type theory which admits partial functions, undefined terms, and subtypes. The IMPS system is available without fee under the terms of a public license.

---

arguments and negation when it has one argument.

[2] In particular, each expression of the form `-`x is replaced with the expression `(- 0` x`)`. As a consequence, we can assume that in Simple Pre-Scheme programs the primitive operator `-` always represents subtraction, and not negation.

Several aspects of IMPS makes it especially well suited as an environment in which to analyze numerical programs:

(1) IMPS supports the little theories version of the axiomatic method [1]. Hence numerical objects can be formalized as members of abstract numerical datatypes, and one formalization can be related to an alternative formalization by means of theory interpretation.

(2) IMPS contains a well developed theory of the real numbers called H-O-Real-Arithmetic (which is essentially the theory of a complete ordered field with the integers and the rational numbers specified as substructures of the real numbers). The theory Machine-Arithmetic (described in section 4) is an extension of H-O-Real-Arithmetic.

(3) In the IMPS logic, types are allowed to have subtypes. (Types and subtypes are called collectively *sorts*.) Hence a numerical datatype can be formalized in IMPS as a subsort of one of the sorts in H-O-Real-Arithmetic. For example, the type of machine integers is formalized in Machine-Arithmetic as a subsort of $\mathbf{Z}$, the sort of integers in H-O-Real-Arithmetic. This allows the objects of a numerical datatype to be thought of as ordinary abstract mathematical objects, such as the real numbers.

(4) IMPS admits partial functions and undefined terms. Hence operations on numerical objects can be formalized in IMPS as partial functions. This reduces questions about overflow to questions about definedness—for which IMPS has special machinery.

(5) In IMPS one can define a system of functions to be the least fixed point (with respect to the subfunction ordering) of a corresponding system of monotone functionals. This notion of mutual recursion is semantically the same as the notion of the `letrec` construction in Pre-Scheme.

4 The IMPS Theory of Machine Arithmetic

Machine-Arithmetic is a formalization in IMPS of the axiomatization of machine integers proposed in [8]. (Appendix A contains a TEX presentation of the IMPS file which defines Machine-Arithmetic.) Machine-Arithmetic is an extension of the IMPS theory H-O-Arithmetic, so that it contains all the ordinary machinery of real arithmetic. Its base language consists of the language of H-O-Arithmetic plus the following two constants of sort **Z** which are intended to denote, respectively, the largest and smallest machine integers:

(1) maxint.

(2) minint.

Its axioms consist of the axioms of H-O-Arithmetic plus the following two statements about maxint and minint:

(1) $0 < \text{maxint}$.

(2) $\text{minint} = -\text{maxint}$.[3]

An atomic sort named int is defined in Machine-Arithmetic to be the set of all integers which lie inclusively between minint and maxint. The sort int is intended to denote the collection of machine integers. Notice that, since minint and maxint are not fully specified, what is a machine integer in Machine-Arithmetic is also not fully specified.

For each standard primitive for machine arithmetic in Simple Pre-Scheme, a corresponding constant is defined in Machine-Arithmetic (see Table 1). These constants are defined to be the ordinary predicates and functions of H-O-Real-Arithmetic restricted to int. Thus, they are undefined outside of int. Several basic lemmas about these constants are proved in the file defining Machine-Arithmetic.

5 The Pre-Scheme-to-IMPS Translator

The Pre-Scheme-to-IMPS Translator is a procedure named `ps-to-imps` which "compiles" a purely functional VLISP Pre-Scheme program P that manipulates machine integers into an IMPS representation consisting of:

(1) An extension T of the theory Machine-Arithmetic (described in section 4) specified by a set of IMPS def-forms. (A *def-form* is a syntactic form for specifying an IMPS entity. For more information, see [3].)

(2) An expression E in T specified by a view-expr form which is intended to represent the body of P.

---

[3] Alternate specifications of minint are given in [8]. We have chosen to specify minint as the negation of maxint because it yields the simplest and most elegant theory of machine integers.

Simple Pre-Scheme Primitive	IMPS Constant
`=`	$=_{\mathrm{ma}}$
`<`	$<_{\mathrm{ma}}$
`<=`	$<=_{\mathrm{ma}}$
`>`	$>_{\mathrm{ma}}$
`>=`	$>=_{\mathrm{ma}}$
`+`	$+_{\mathrm{ma}}$
`*`	$*_{\mathrm{ma}}$
`-`	$\mathrm{sub}_{\mathrm{ma}}$
`abs`	$\mathrm{abs}_{\mathrm{ma}}$
`quotient`	$\mathrm{div}_{\mathrm{ma}}$
`remainder`	$\mathrm{mod}_{\mathrm{ma}}$

Table 1: Defined Constants in Machine-Arithmetic.

More precisely, `ps-to-imps` takes two arguments:

(1) An input file containing the Pre-Scheme program P.

(2) An output file in which is put the list of def-forms that specify T and E.

We believe that, taken together the IMPS theory T and the expression E *faithfully* represent the Pre-Scheme program P, but we have not written down a proof of this claim.

The final product of `ps-to-imps`, the list of def-forms placed in the output file, is generated in three stages. The first stage translates P into a Simple Pre-Scheme program P'. This stage is exactly the same as the first stage of the VLISP Pre-Scheme compiler. The second stage extracts from P' a certain list L of information. Then the third stage translates L into the list D of def-forms given in the IMPS sexp (s-expression) syntax. `ps-to-imps` is written in the T programming language, but some of its subprocedures are written in Scheme. In particular, the first two stages of `ps-to-imps` are performed by procedures written in Scheme.

D contains, in order, the following def-forms:

(1) A def-language form that defines a language named Machine-Arithmetic-Language-Extension. This language contains the language of theory Machine-Arithmetic plus the following constants:

   (a) A constant with a name of the form $\_\mathrm{unspecified}_n$ for each expression in P' of the form `(if #f #f)` or `(set! I E)`.
   (b) A constant of sort int with the name $\mathrm{zero}_{\mathrm{ma}}$, $\mathrm{plus\_}m_{\mathrm{ma}}$, or $\mathrm{minus\_}n_{\mathrm{ma}}$ for each numerical constant `0`, `-`m, or n, respectively, in P'.

(2) A def-theory form that defines a theory named Machine-Arithmetic-Language-Extension. This theory contains the language Machine-Arithmetic-Language-Extension, the theory Machine-Arithmetic, and the following axioms:

   (a) $\mathrm{zero}_{\mathrm{ma}} = 0$ if $\mathrm{zero}_{\mathrm{ma}}$ is a constant of Machine-Arithmetic-Language-Extension.
   (b) $\mathrm{plus\_}m_{\mathrm{ma}} = m$ if $\mathrm{plus\_}m_{\mathrm{ma}}$ is a constant of Machine-Arithmetic-Language-Extension.
   (c) $\mathrm{minus\_}n_{\mathrm{ma}} = -n$ if $\mathrm{minus\_}n_{\mathrm{ma}}$ is a constant of Machine-Arithmetic-Language-Extension.

   Notice that these axioms constrain the values that the primitive constants minint and maxint may have.

(3) A def-recursive-constant form corresponding to the bindings of the single `letrec` expression in P'.

(4) A view-expr form corresponding to the body of the single `letrec` expression in P'.

6 Examples

This section contain five examples of machine integer Pre-Scheme programs that were analyzed using the machinery described in the previous sections. Each example was carried out as follows. A VLISP Pre-Scheme program was written in a file `program.scm`. Next the following command was executed in a UNIX shell:

```
ps-to-imps program.scm program.t
```

Then a formal analysis of the program was performed in IMPS within the theory specified by the def-forms that were put in `program.t`. And, finally, the def-forms and results of the analysis (i.e., definitions, theorems, proofs, etc.) were placed in an IMPS file. Appendices B, C, D, E, and F contain a TeX presentation of the IMPS file for each example, respectively.

6.1 Even and Odd Testers

6.1.1 The Pre-Scheme Program

```
(define (even-nn x) (if (zero? x) 1 (odd-nn (- x 1))))

(define (odd-nn x) (if (zero? x) 0 (even-nn (- x 1))))

(even-nn 77)
```

6.1.2 Discussion

This program defines by mutual recursion two procedures `even-nn` and `odd-nn` which test for whether a natural number (represented as a machine integer) is even and odd, respectively. When the test succeeds, `1` is returned, and when the test fails, `0` is returned. The final command tests whether `77` is even. Notice that `even-nn` and `odd-nn` returns an overflow error when they are applied to a negative machine integer.

In our analysis of the program, we prove that:

(1) `(even-nn` n`)` terminates without an error iff n is an nonnegative machine integer.

(2) `(odd-nn` n`)` terminates without an error iff n is an nonnegative machine integer.

(3) `(even-nn` n`)` returns `1` iff n is an even nonnegative machine integer.

(4) `(odd-nn` n`)` returns `1` iff n is an odd nonnegative machine integer.

(5) `(even-nn` n`)` returns `1` iff `(odd-nn` n`)` returns `0`.

(6) `(odd-nn` n`)` returns `1` iff `(even-nn` n`)` returns `0`.

6.2 Recursive Factorial Function

6.2.1 The Pre-Scheme Program

```
(define (fact n)
  (if (zero? n)
      1
      (if (positive? n)
      (* n (fact (- n 1))))))

(fact 4)
```

6.2.2 Discussion

This program defines by direct recursion a procedure `fact` which computes the factorial function on the natural numbers (represented as machine integers). The final command computes the factorial of `4`. Notice that `fact` returns an overflow error when it is applied to a negative machine integer.

In our analysis of the program, we prove that, if `(fact` n`)` terminates without an error, then it returns $n!$.

6.3 Iterative Factorial Function

6.3.1 The Pre-Scheme Program

```
(define (fact-loop n a)
  (if (positive? n)
      (fact-loop (- n 1) (* n a))
      a))

(define-integrable (fact n)
  (fact-loop n 1))

(fact 4)
```

6.3.2 Discussion

This program defines by tail recursion a procedure `fact` which computes the factorial function on the natural numbers (represented as machine integers). (Since `fact` is defined by tail recursion, it executes in constant space; i.e., it is an iterative procedure.) The final command computes the factorial of

4. Notice that `fact` returns `1` when it is applied to a nonpositive machine integer.

In our analysis of the program, we prove that, if (`fact-loop` n a) terminates without an error, then it returns $(n!) * a$.

6.4 Fibonacci Function

6.4.1 The Pre-Scheme Program

```
(define (fib-1 n)
  (if (= n 0)
      1
      (+ (fib-1  (- n 1)) (fib-2 (- n 1)))))

(define (fib-2 n)
  (if (= n 0)
      0
      (fib-1 (- n 1))))

(fib-1 1)
```

6.4.2 Discussion

This program defines by mutual recursion two procedures `fib-1` and `fib-2`. The procedure `fib-1` computes the fibonacci sequence on the natural numbers represented as machine integers. We prove that, if (`fib-1` n) terminates without an error, then it returns the nth fibonacci number.

6.5 Greatest Common Denominator

6.5.1 The Pre-Scheme Program

```
(define (gcd_scm u v)
  (if (and (<= 0 v) (<= 0 u))
      (if (= u 0)
     v
     (if (= v 0)
         u
         (gcd_scm v (remainder u v))))))

(gcd_scm 6 7)
```

6.5.2 Discussion

This example illustrates the strong interplay between abstract mathematical concepts and concrete numerical programs. The program defines a procedure `gcd_scm` which computes the greatest common divisor of a pair of machine integers. In our analysis of the program we prove that, if `(gcd_scm` m n`)` terminates without error whenever m, n are nonnegative machine integers, and that it returns the greatest common divisor of m, n.

The greatest common divisor of two integers a, b is defined as the unique positive generator of the set of integer combinations $ra + sb$. A generator of a set is an element c of the set such that every other element is a multiple (positive or negative) of c, and moreover, the set contains only multiples of c. This definition is adopted because it is closer to the traditional mathematical approach which defines divisibility in terms of ideals. We prove in IMPS that this definition is equivalent with a number of other characterizations, including the characterization as the largest positive integer which divides both a and b.

7 Conclusion

We believe that the work described in this paper is a good first step toward a useful system for formally analyzing numerical programs. The following are the main advantages of our method:

- The translation from Pre-Scheme to IMPS is completely automated.
- The formal analysis of a numerical Pre-Scheme program is performed using IMPS, a state-of-the-art theorem proving system.
- The numerical datatypes are represented directly as "subtypes" of ordinary abstract mathematical datatypes such as real number arithmetic. This makes it possible to apply the results of traditional mathematics wherever they are needed.
- Questions about overflow in numerical Pre-Scheme programs are reduced to questions about the definedness of IMPS expressions. IMPS has been specifically designed to facilitate reasoning about definedness.
- After a Pre-Scheme numerical program is analyzed and shown to be correct, it can be translated into either C or assembly language. More-

over, the software that translates Pre-Scheme into assembly language has been verified (see [7]).

- Although the examples we tested were very simple, the method will scale up to larger and more complex examples.

There are also a couple of fairly obvious disadvantages of our method:

- The object programs must be written in Pre-Scheme.
- The Pre-Scheme programs must be purely functional, i.e., they are not allowed to modify the values of variables.

We think that, for the sake of better software assurance, many software developers would be willing to live with these disadvantages, especially since Pre-Scheme is very similar to C.

A major disadvantage of the implementation of our method is that it cannot handle Pre-Scheme programs that manipulate floating-point reals. Since most interesting and commonly used numerical programs do manipulate floating-point reals, our implementation would be much more useful if it could support floating-point reals. Our plan for the future is extend our implementation to handle Pre-Scheme programs that manipulate both machine integers and floating-point reals. Fortunately, there is no conceptual obstacle to this task. However, it is a nontrivial task since the numerical datatype of floating-point reals is much more complicated than the numerical datatype of machine integers.

The task would involve three subtasks:

- Add new primitive operators to VLISP Pre-Scheme for handling floating-point reals.
- Formulate an IMPS theory of floating-point arithmetic (that would be an extension of Machine-Arithmetic), closely following the Payne-Schaffert-Wichmann proposed standard [8].
- Extend the Pre-Scheme-to-IMPS translator to handle purely functional VLISP Pre-Scheme programs that manipulate both machine integers and floating-point reals.

Most of the work will be involved in the first two subtasks; the last subtask will be relatively easy.

Once the implementation is ready to handle floating-point reals, software developers will have a very useful tool for formally analyzing numerical programs. As far as we know, there is no comparable tool available today.

Component theory: h-o-real-arithmetic
Top level axioms:

maxint-is-positive $0 < \text{maxint}$.

minint-is-negative-maxint $\text{minint} = -\text{maxint}$.

Figure 1: Components and axioms for machine-arithmetic

A The File for Machine-Arithmetic

```
(load-section number-theory)

(include-files
 (files
  (imps /theories/machine-arithmetic/ma-presentation)))
```

Language A.1 (mach-arith-language)
Embedded language: *h-o-real-arithmetic*
Constants: maxint *:* **Z**
minint *:* **Z**

Theory A.2 (machine-arithmetic)
Language: *mach-arith-language*
Component Theories and Axioms: *See Figure 1.*

Theorem A.3 (minint-is-negative)
Theory: machine-arithmetic
$\text{minint} < 0$.

```
(proof
  (

   (apply-macete-with-minor-premises minint-is-negative-maxint)
   simplify

   ))
```

Sort Definition A.4 (int)
Theory: machine-arithmetic
$[\, i : \mathbf{Z} \;\mapsto$
conjunction
- minint $\leq i$
- $i \leq$ maxint $]$.

Definition A.5 (=_ma)
Theory: machine-arithmetic
$[\, x, y : \text{int} \;\mapsto$
$x = y\,]$.

Definition A.6 (¡_ma)
Theory: machine-arithmetic
$[\, x, y : \text{int} \;\mapsto$
$x < y\,]$.

Definition A.7 (¡=_ma)
Theory: machine-arithmetic
$[\, x, y : \text{int} \;\mapsto$
$x \leq y\,]$.

Definition A.8 (¿_ma)
Theory: machine-arithmetic
$[\, x, y : \text{int} \;\mapsto$
$x > y\,]$.

Definition A.9 (¿=_ma)
Theory: machine-arithmetic
$[\, x, y : \text{int} \;\mapsto$
$x \geq y\,]$.

```
(def-script closed-on-int-2 0
  (
   sort-definedness
   direct-inference
   (case-split ("#(xx_0,int) and #(xx_1,int)"))
   simplify
   (simplify-antecedent "with(p:prop,p);")
   ))
```

```
(def-script unfold-ma-defined-expression 1
  (
   direct-inference
   (unfold-single-defined-constant-globally $1)
   (case-split ("#(x,int) and #(y,int)"))
   simplify
   (simplify-antecedent "with(p:prop,p);")
   ))
```

Lemma A.10 (Anonymous-14)
Theory: machine-arithmetic
$[\,x, y : \text{int} \;\mapsto$
conditionally, if $x + y \downarrow \text{int}$

- *then* $x + y$
- *else* $\bot\text{int}\,] \downarrow [\text{int} \times \text{int} \rightharpoonup \text{int}]$.

Definition A.11 (+_ma)
Theory: machine-arithmetic
$[\,x, y : \text{int} \;\mapsto$
conditionally, if $x + y \downarrow \text{int}$

- *then* $x + y$
- *else* $\bot\text{int}\,]$.

Theorem A.12 (unfold-defined-expression%+_ma)
Theory: machine-arithmetic
$\forall x, y : \mathbf{Z} \quad s.\ t. \quad +_{\text{ma}}(x, y) \downarrow,$
$+_{\text{ma}}(x, y) = x + y.$

```
(proof
 (

  (unfold-ma-defined-expression +_ma)

  ))
```

Lemma A.13 (Anonymous-15)
Theory: machine-arithmetic
$[\,x, y : \text{int} \;\mapsto$
conditionally, if $x \cdot y \downarrow \text{int}$

- *then* $x \cdot y$
- *else* $\bot\text{int}\,] \downarrow [\text{int} \times \text{int} \rightharpoonup \text{int}]$.

Definition A.14 (*_ma)
Theory: machine-arithmetic
$[\,x, y : \mathrm{int} \mapsto$
conditionally, if $x \cdot y \downarrow \mathrm{int}$
- *then* $x \cdot y$
- *else* $\bot\mathrm{int}\,]$.

Theorem A.15 (unfold-defined-expression%*_ma)
Theory: machine-arithmetic
$\forall x, y : \mathbf{Z}$ *s. t.* $*_{\mathrm{ma}}\,(x, y) \downarrow$,
$*_{\mathrm{ma}}\,(x, y) = x \cdot y$.

```
(proof
  (

   (unfold-ma-defined-expression *_ma)

   ))
```

Lemma A.16 (Anonymous-16)
Theory: machine-arithmetic
$[\,x, y : \mathrm{int} \mapsto$
conditionally, if $x - y \downarrow \mathrm{int}$
- *then* $x - y$
- *else* $\bot\mathrm{int}\,] \downarrow [\mathrm{int} \times \mathrm{int} \rightharpoonup \mathrm{int}]$.

Definition A.17 (sub_ma)
Theory: machine-arithmetic
$[\,x, y : \mathrm{int} \mapsto$
conditionally, if $x - y \downarrow \mathrm{int}$
- *then* $x - y$
- *else* $\bot\mathrm{int}\,]$.

Theorem A.18 (unfold-defined-expression%sub_ma)
Theory: machine-arithmetic
$\forall x, y : \mathbf{Z}$ *s. t.* $\mathrm{sub}_{\mathrm{ma}}(x, y) \downarrow$,
$\mathrm{sub}_{\mathrm{ma}}(x, y) = x - y$.

```
(proof
  (

   (unfold-ma-defined-expression sub_ma)

   ))
```

Lemma A.19 (unary-int-function-lemma)
Theory: machine-arithmetic
$\forall f : \mathbf{R} \rightharpoonup \mathbf{R}$ implication

- $\forall x : \mathbf{Z}$ conjunction
  - $|f(x)| \leq |x|$
  - implication
    - $f(x) \downarrow$
    - $f(x) \downarrow \mathbf{Z}$
- $[\, x : \mathrm{int} \mapsto$
  $f(x) \,] \downarrow [\mathrm{int} \rightharpoonup \mathrm{int}]$.

Lemma A.20 (binary-int-function-lemma)
Theory: machine-arithmetic
$\forall f : \mathbf{R} \times \mathbf{R} \rightharpoonup \mathbf{R}$ implication

- $\forall x, y : \mathbf{Z}$ conjunction
  - $|f(x, y)| \leq |x|$
  - implication
    - $f(x, y) \downarrow$
    - $f(x, y) \downarrow \mathbf{Z}$
- $[\, x, y : \mathrm{int} \mapsto$
  $f(x, y) \,] \downarrow [\mathrm{int} \times \mathrm{int} \rightharpoonup \mathrm{int}]$.

Theorem A.21 (int-minus-lemma)
Theory: machine-arithmetic
$\forall x : \mathrm{int} \quad -x \downarrow \mathrm{int}$.

```
(proof
  (


   direct-and-antecedent-inference-strategy
   (cut-with-single-formula "#(x,int)")
   (incorporate-antecedent "with(p:prop,p);")
   (apply-macete-with-minor-premises
    int-defining-axiom_machine-arithmetic)
   (apply-macete-with-minor-premises minint-is-negative-maxint)
   simplify

   ))
```

Lemma A.22 (Anonymous-17)
Theory: machine-arithmetic
$[\, x : \mathrm{int} \mapsto$
$-x \,] \downarrow [\mathrm{int} \rightharpoonup \mathrm{int}]$.

Definition A.23 (-_ma)
Theory: machine-arithmetic
$[\, x : \mathrm{int} \;\mapsto$
$\quad -x \,].$

Lemma A.24 (Anonymous-18)
Theory: machine-arithmetic
$[\, x : \mathrm{int} \;\mapsto$
$\quad |x| \,] \downarrow [\mathrm{int} \rightharpoonup \mathrm{int}].$

Definition A.25 (abs_ma)
Theory: machine-arithmetic
$[\, x : \mathrm{int} \;\mapsto$
$\quad |x| \,].$

Theorem A.26 (maxint-division-lemma)
Theory: machine-arithmetic
$\forall a : \mathrm{int}, b : \mathbf{Z} \quad s.\ t. \quad \neg(b = 0),$
$\quad a/b \leq \mathrm{maxint}.$

```
(proof
 (

  (cut-with-single-formula
   "forall(a:int,b:zz,0<b implies a/b<=maxint)")
  (block
   (script-comment "`cut-with-single-formula' at (0)")
   direct-and-antecedent-inference-strategy
   (case-split ("0<b"))
   simplify
   (block
    (script-comment "`case-split' at (2)")
    (force-substitution "a/b" "(-a)/(-b)" (0))
    (block
     (script-comment "`force-substitution' at (0)")
     (backchain "with(p:prop,forall(a:int,b:zz,p));")
     simplify
     (block
      (script-comment "`backchain' at (1)")
      (cut-with-single-formula "#(a,int)")
      (incorporate-antecedent "with(a:int,#(a,int));")
      (apply-macete-with-minor-premises
       int-defining-axiom_machine-arithmetic)
      (apply-macete-with-minor-premises minint-is-negative-maxint)
```

```
       simplify))
     simplify))
   (block
    (script-comment "‘cut-with-single-formula’ at (1)")
    direct-and-antecedent-inference-strategy
    (apply-macete-with-minor-premises
     fractional-expression-manipulation)
    (cut-with-single-formula "maxint<=maxint*b")
    (move-to-sibling 1)
    (block
     (script-comment "‘cut-with-single-formula’ at (1)")
     (cut-with-single-formula "0<=maxint*(b-1)")
     simplify
     simplify)
    (block
     (script-comment "‘cut-with-single-formula’ at (0)")
     (cut-with-single-formula "a<=maxint")
     simplify
     (block
      (script-comment "‘cut-with-single-formula’ at (1)")
      (cut-with-single-formula "minint <= a and a <= maxint")
      (apply-macete-with-minor-premises
       int-in-quasi-sort_machine-arithmetic))))

   ))
```

Theorem A.27 (minint-division-lemma)

Theory: machine-arithmetic

$\forall a : \mathrm{int}, b : \mathbf{Z} \quad s.\ t. \quad \neg(b = 0),$
$\quad \mathrm{minint} \leq a/b.$

```
(proof
  (

   direct-and-antecedent-inference-strategy
   (apply-macete-with-minor-premises minint-is-negative-maxint)
   (cut-with-single-formula "(-a)/b<=maxint")
   simplify
   (block
    (script-comment
     "node added by ‘cut-with-single-formula’ at (1)")
    (apply-macete-with-minor-premises maxint-division-lemma)
    (cut-with-single-formula "#(a,int)")
    (incorporate-antecedent "with(a:int,#(a,int));")
    (apply-macete-with-minor-premises
     int-defining-axiom_machine-arithmetic)
    (apply-macete-with-minor-premises minint-is-negative-maxint)
```

```
  simplify)

 ))
```

Theorem A.28 (int-division-lemma)
Theory: machine-arithmetic
$\forall a : \mathrm{int}, b : \mathbf{Z} \quad s.\ t. \quad \neg(b = 0),$
$\quad \mathrm{div}(a, b) \downarrow \mathrm{int}.$

```
(proof
 (

  direct-and-antecedent-inference-strategy
  (apply-macete-with-minor-premises
   int-defining-axiom_machine-arithmetic)
  beta-reduce-repeatedly
  (unfold-single-defined-constant-globally div)
  (apply-macete-with-minor-premises floor-not-much-below-arg)
  direct-and-antecedent-inference-strategy
  (apply-macete-with-minor-premises minint-division-lemma)
  (block
   (script-comment
    "‘direct-and-antecedent-inference-strategy’ at (1)")
   (cut-with-single-formula "floor(a/b)<=a/b and a/b<=maxint")
   simplify
   (block
    (script-comment "‘cut-with-single-formula’ at (1)")
    (apply-macete-with-minor-premises floor-below-arg)
    (apply-macete-with-minor-premises maxint-division-lemma)))

 ))
```

Lemma A.29 (Anonymous-19)
Theory: machine-arithmetic
$[\, x, y : \mathrm{int} \;\mapsto$
$\quad \mathrm{div}(x, y)\,] \downarrow [\mathrm{int} \times \mathrm{int} \rightharpoonup \mathrm{int}].$

Definition A.30 (div_ma)
Theory: machine-arithmetic
$[\, x, y : \mathrm{int} \;\mapsto$
$\quad \mathrm{div}(x, y)\,].$

Theorem A.31 (maxint-pos-mod-lemma)
Theory: machine-arithmetic

$\forall a : \mathbf{Z}, b : \mathrm{int} \quad s.\ t. \quad 0 < b,$
$\quad \mathrm{mod}(a, b) < \mathrm{maxint}.$

```
(proof
  (

   direct-and-antecedent-inference-strategy
   (cut-with-single-formula " a mod b < b and b<=maxint")
   simplify
   (block
    (script-comment "‘cut-with-single-formula’ at (1)")
    (cut-with-single-formula "#(b,int)")
    (incorporate-antecedent "with(b:int,#(b,int));")
    (apply-macete-with-minor-premises
     int-defining-axiom_machine-arithmetic)
    beta-reduce-repeatedly
    direct-and-antecedent-inference-strategy
    (instantiate-theorem division-with-remainder ("b" "a")))

   ))
```

Theorem A.32 (minint-pos-mod-lemma)

Theory: machine-arithmetic
$\forall a : \mathbf{Z}, b : \mathrm{int} \quad s.\ t. \quad 0 < b,$
$\quad \mathrm{minint} < \mathrm{mod}(a, b).$

```
(proof
  (

   direct-and-antecedent-inference-strategy
   (cut-with-single-formula "0<= a mod b")
   simplify
   (instantiate-theorem division-with-remainder ("b" "a"))

   ))
```

Theorem A.33 (int-mod-lemma)

Theory: machine-arithmetic
$\forall a : \mathbf{Z}, b : \mathrm{int} \quad s.\ t. \quad \neg(b = 0),$
$\quad \mathrm{mod}(a, b) \downarrow \mathrm{int}.$

```
(proof
  (

   (cut-with-single-formula
```

```
 "forall(a:zz,b:int,0<b implies #(a mod b,int))")
(block
 (script-comment "‘cut-with-single-formula’ at (0)")
 direct-and-antecedent-inference-strategy
 (case-split ("0<b"))
 simplify
 (block
  (script-comment "‘case-split’ at (2)")
  (force-substitution "a mod b" "-(- a mod -b)" (0))
  (block
   (script-comment "‘force-substitution’ at (0)")
   (apply-macete-with-minor-premises int-minus-lemma)
   (backchain "with(p:prop,forall(a:zz,b:int,p));")
   simplify
   (block
    (script-comment "‘backchain’ at (1)")
    (cut-with-single-formula "#(-b, int)")
    (simplify-antecedent "with(r:rr,#(r,int));")
    (apply-macete-with-minor-premises int-minus-lemma)))
  (block
   (script-comment "‘force-substitution’ at (1)")
   (apply-macete-with-minor-premises mod-of-negative)
   simplify)))
(block
 (script-comment "‘cut-with-single-formula’ at (1)")
 direct-and-antecedent-inference-strategy
 (cut-with-single-formula "#(a mod b ,zz)")
 (block
  (script-comment "‘cut-with-single-formula’ at (0)")
  (apply-macete-with-minor-premises
   int-defining-axiom_machine-arithmetic)
  beta-reduce-repeatedly
  simplify
  (cut-with-single-formula "minint<a mod b and a mod b<maxint")
  simplify
  (block
   (script-comment "‘cut-with-single-formula’ at (1)")
   (apply-macete-with-minor-premises minint-pos-mod-lemma)
   (apply-macete-with-minor-premises maxint-pos-mod-lemma)))
 (block
  (script-comment "‘cut-with-single-formula’ at (1)")
  (apply-macete-with-minor-premises mod-of-integer-is-integer)
  simplify))

))
```

Lemma A.34 (Anonymous-20)
Theory: machine-arithmetic
$[\,x, y : \mathrm{int} \mapsto$
$\quad \mathrm{mod}(x, y)\,] \downarrow [\mathrm{int} \times \mathrm{int} \rightharpoonup \mathrm{int}].$

Definition A.35 (mod_ma)
Theory: machine-arithmetic
$[\,x, y : \mathrm{int} \mapsto$
$\quad \mathrm{mod}(x, y)\,].$

B The File for Even and Odd Testers

```
(include-files
 (files
  (imps /theories/machine-arithmetic/machine-arithmetic)))
```

Language B.1 (machine-arithmetic-language-extension)
Embedded language: *machine-arithmetic*
Constants: $\mathrm{zero}_{\mathrm{ma}}$ *:* int
$\mathrm{plus\_1}_{\mathrm{ma}}$ *:* int
$\mathrm{minus\_1}_{\mathrm{ma}}$ *:* int
$\mathrm{\_unspecified}_0$ *:* int
$\mathrm{\_unspecified}_1$ *:* int
$\mathrm{\_unspecified}_2$ *:* int
$\mathrm{\_unspecified}_3$ *:* int
$\mathrm{plus\_77}_{\mathrm{ma}}$ *:* int

Theory B.2 (machine-arithmetic-extension)
Language: *machine-arithmetic-language-extension*
Component Theories and Axioms: *See Figure 2.*

The following 2 definitions are mutually recursive.

Definition (Recursive) B.3 (odd%nn)
Theory: machine-arithmetic-extension
$[\,\mathrm{odd\_nn}, \mathrm{even\_nn} : \mathrm{int} \rightharpoonup \mathrm{int} \mapsto$
$\quad [\,\mathrm{x1} : \mathrm{int} \mapsto$

Component theory: machine-arithmetic
Top level axioms:

machine-arithmetic-extension-axiom-0 $\mathrm{zero}_{\mathrm{ma}} = 0.$

machine-arithmetic-extension-axiom-1 $\mathrm{plus\_1}_{\mathrm{ma}} = 1.$

machine-arithmetic-extension-axiom-2 $\mathrm{minus\_1}_{\mathrm{ma}} = -1.$

machine-arithmetic-extension-axiom-3 $\mathrm{plus\_77}_{\mathrm{ma}} = 77.$

Figure 2: Components and axioms for machine-arithmetic-extension

conditionally, if $=_{\mathrm{ma}} (\mathrm{zero}_{\mathrm{ma}}, \mathrm{x1})$
- *then* $\mathrm{zero}_{\mathrm{ma}}$
- *else* $\mathrm{even\_nn}(+_{\mathrm{ma}}(\mathrm{minus\_1}_{\mathrm{ma}}, \mathrm{x1}))\,]\,].$

Definition (Recursive) B.4 (even%nn)
Theory: machine-arithmetic-extension
$[\,\mathrm{odd\_nn}, \mathrm{even\_nn} : \mathrm{int} \rightharpoonup \mathrm{int} \mapsto$
$[\,x : \mathrm{int} \mapsto$
conditionally, if $=_{\mathrm{ma}} (\mathrm{zero}_{\mathrm{ma}}, x)$
- *then* $\mathrm{plus\_1}_{\mathrm{ma}}$
- *else* $\mathrm{odd\_nn}(+_{\mathrm{ma}}(\mathrm{minus\_1}_{\mathrm{ma}}, x))\,]\,].$

```
(view-expr "(apply-operator even%nn plus%77_ma)"
  (language machine-arithmetic-extension)
  (syntax sexp-syntax))

(def-compound-macete rewrite-integer-constants
  (series
   machine-arithmetic-extension-axiom-0
   machine-arithmetic-extension-axiom-1
   machine-arithmetic-extension-axiom-2
   machine-arithmetic-extension-axiom-3))
```

Theorem B.5 (even%nn-odd%nn-definedness-lemma-1)
Theory: machine-arithmetic-extension

$\forall n : \mathbf{Z} \quad s.\ t. \quad 0 \leq n \land n \downarrow \mathrm{int},$
conjunction
- $\mathrm{even\_nn}(n) \downarrow$
- $\mathrm{odd\_nn}(n) \downarrow .$

```
(proof
  (

    (induction trivial-integer-inductor ("n"))
    (block
     (script-comment "‘induction’ at (0 0 0 0 0 0 0 0)")
     unfold-defined-constants
     (unfold-single-defined-constant-globally =_ma)
     (apply-macete-with-minor-premises rewrite-integer-constants))
    (block
     (script-comment
      "‘induction’ at (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)")
     (unfold-single-defined-constant-globally =_ma)
     (apply-macete-with-minor-premises rewrite-integer-constants)
     (unfold-single-defined-constant-globally +_ma)
     simplify
     (incorporate-antecedent "with(r:rr,#(r,int));")
     (apply-macete-with-minor-premises
      int-defining-axiom_machine-arithmetic)
     simplify)

    ))
```

Theorem B.6 (even%nn-odd%nn-definedness-lemma-2)

Theory: machine-arithmetic-extension
$\forall n : \mathbf{Z} \quad s.\ t. \quad n < 0,$
conjunction
- $\neg(\mathrm{even\_nn}(n) \downarrow)$
- $\neg(\mathrm{odd\_nn}(n) \downarrow).$

```
(proof
  (

    direct-inference
    direct-inference
    (instantiate-theorem
     program-letrec-strong-minimality_machine-arithmetic-extension
     ("lambda(x:int,if(0<=x,odd%nn(x),?int))"
      "lambda(x:int,if(0<=x,even%nn(x),?int))"))
    (block
     (script-comment "‘instantiate-theorem’ at (0 0 0 0 0)")
```

```
 (contrapose "with(p:prop,not(p));")
 (unfold-single-defined-constant-globally odd%nn)
 (case-split ("zero_ma =_ma u_0"))
 (block
  (script-comment "‘case-split’ at (1)")
  simplify
  (incorporate-antecedent "with(u_0:int,zero_ma =_ma u_0);")
  (unfold-single-defined-constant-globally =_ma)
  (apply-macete-with-minor-premises rewrite-integer-constants)
  simplify)
 (block
  (script-comment "‘case-split’ at (2)")
  (incorporate-antecedent "with(i:int,#(i));")
  simplify
  direct-inference
  (cut-with-single-formula
   "0=0 and #(lambda(x:int,if(0<=x, even%nn(x), ?int))
 (minus%1_ma +_ma u_0))")
  (incorporate-antecedent "with(i:int,#(i));")
  beta-reduce-repeatedly
  simplify
  direct-and-antecedent-inference-strategy
  (incorporate-antecedent "with(i:int,0<=i);")
  (apply-macete-with-minor-premises unfold-defined-expression%+_ma)
  (apply-macete-with-minor-premises rewrite-integer-constants)
  simplify))
(block
 (script-comment "‘instantiate-theorem’ at (0 0 1 0 0)")
 (contrapose "with(p:prop,not(p));")
 (unfold-single-defined-constant-globally even%nn)
 (case-split ("zero_ma =_ma u_0"))
 (block
  (script-comment "‘case-split’ at (1)")
  simplify
  (incorporate-antecedent "with(u_0:int,zero_ma =_ma u_0);")
  (unfold-single-defined-constant-globally =_ma)
  (apply-macete-with-minor-premises rewrite-integer-constants)
  simplify)
 (block
  (script-comment "‘case-split’ at (2)")
  (incorporate-antecedent "with(i:int,#(i));")
  simplify
  direct-inference
  (cut-with-single-formula
   "0=0 and #(lambda(x:int,if(0<=x, odd%nn(x), ?int))
 (minus%1_ma +_ma u_0))")
  (incorporate-antecedent "with(i:int,#(i));")
```

```
  beta-reduce-repeatedly
  simplify
  direct-and-antecedent-inference-strategy
  (incorporate-antecedent "with(i:int,0<=i);")
  (apply-macete-with-minor-premises
   unfold-defined-expression%+_ma)
  (apply-macete-with-minor-premises rewrite-integer-constants)
  simplify))
(block
 (script-comment "`instantiate-theorem' at (0 1 0)")
 (case-split ("#(n,int)"))
 (block
  (script-comment "`case-split' at (1)")
  direct-inference
  (block
   (script-comment "`direct-inference' at (0)")
   (instantiate-universal-antecedent
    "with(f:[int,int],
       forall(u_0:int,
         #(even%nn(u_0)) implies even%nn(u_0)=f(u_0)));"
    ("n"))
   (incorporate-antecedent "with(i:int,i=i);")
   simplify)
  (block
   (script-comment "`direct-inference' at (1)")
   (instantiate-universal-antecedent
    "with(f:[int,int],
       forall(u_0:int,#(odd%nn(u_0)) implies odd%nn(u_0)=f(u_0)));"
    ("n"))
   (incorporate-antecedent "with(i:int,i=i);")
   simplify))
 (block
  (script-comment "`case-split' at (2)")
  direct-inference
  (contrapose "with(p:prop,not(p));")
  (contrapose "with(n:zz,not(#(n,int)));")))

))
```

Theorem B.7 (even%nn-odd%nn-definedness)

Theory: machine-arithmetic-extension

$\forall n : \mathbf{Z} \quad \iff$

- conjunction
  - even_nn$(n)\downarrow$
  - odd_nn$(n)\downarrow$
- conjunction

  - ○ $0 \leq n$
  - ○ $n \downarrow \text{int}$.

```
(proof
  (

   direct-inference
   direct-inference
   (block
    (script-comment "‘direct-inference’ at (0)")
    (instantiate-theorem even%nn-odd%nn-definedness-lemma-2 ("n"))
    simplify)
   (apply-macete-with-minor-premises
    even%nn-odd%nn-definedness-lemma-1)

   ))
```

Theorem B.8 (correctness-of-even%nn-odd%nn-lemma-1)
Theory: machine-arithmetic-extension
$\forall i : \mathbf{Z} \quad s.\ t. \quad 0 \leq i \wedge i \downarrow \text{int}$,
conjunction

- $\Longleftrightarrow$
  - ○ $\text{even\_nn}(i) = 1$
  - ○ $\exists j : \text{int} \quad i = 2 \cdot j$
- $\Longleftrightarrow$
  - ○ $\text{odd\_nn}(i) = 1$
  - ○ $\exists j : \text{int} \quad i = 2 \cdot j + 1$.

```
(proof
  (

   (induction trivial-integer-inductor ("i"))
   (block
    (script-comment "‘induction’ at (0 0 0 0 0 0 0 0)")
    beta-reduce-repeatedly
    direct-and-antecedent-inference-strategy
    (block
     (script-comment
      "‘direct-and-antecedent-inference-strategy’ at (0 0 0)")
     (instantiate-existential ("0"))
     simplify)
    (block
     (script-comment
      "‘direct-and-antecedent-inference-strategy’ at (0 0 1 0)")
     (weaken (0))
```

```
  (unfold-single-defined-constant-globally even%nn)
  (unfold-single-defined-constant-globally =_ma)
  (apply-macete-with-minor-premises rewrite-integer-constants))
 (block
  (script-comment
   "‘direct-and-antecedent-inference-strategy’ at (0 1 0 0 0)")
  (contrapose "odd%nn(0)=1;")
  (weaken (2 1 0))
  (unfold-single-defined-constant-globally odd%nn)
  (unfold-single-defined-constant-globally =_ma)
  (apply-macete-with-minor-premises rewrite-integer-constants))
 (contrapose "with(p:prop,not(p));")
 (block
  (script-comment
   "‘direct-and-antecedent-inference-strategy’ at (0 1 1 0 0 0)")
  (contrapose "with(r:rr,0=r+1);")
  (cut-with-single-formula "j<0 or j=0 or 0<j")
  (block
   (script-comment "‘cut-with-single-formula’ at (0)")
   (antecedent-inference "with(p:prop,p or p or p);")
   simplify
   simplify
   simplify)
  simplify)
 (contrapose "with(p:prop,not(p));"))
(block
 (script-comment
  "‘induction’ at (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)")
 (unfold-single-defined-constant-globally =_ma)
 (unfold-single-defined-constant-globally +_ma)
 (apply-macete-with-minor-premises rewrite-integer-constants)
 simplify
 (backchain "with(p:prop,p implies p);")
 (backchain "with(p:prop,p implies p);")
 direct-and-antecedent-inference-strategy
 (block
  (script-comment
   "‘direct-and-antecedent-inference-strategy’ at (0)")
  (incorporate-antecedent "with(r:rr,#(r,int));")
  (apply-macete-with-minor-premises
   int-defining-axiom_machine-arithmetic)
  simplify)
 (block
  (script-comment
   "‘direct-and-antecedent-inference-strategy’ at (1 1 0 0 0 0)")
  (backchain "with(r:rr,t:zz,t=r);")
  (instantiate-existential ("j+1"))
```

```
    simplify
    (block
     (script-comment "‘instantiate-existential’ at (1 0)")
     (incorporate-antecedent "with(t:zz,#(t,int));")
     (backchain "with(r:rr,t:zz,t=r);")
     (apply-macete-with-minor-premises
      int-defining-axiom_machine-arithmetic)
     simplify))
   (block
    (script-comment
     "‘direct-and-antecedent-inference-strategy’ at (1 1 0 1 1 0)")
    (instantiate-existential ("j-1"))
    simplify
    (block
     (script-comment "‘instantiate-existential’ at (1 0)")
     (incorporate-antecedent "with(t:zz,#(1+t,int));")
     (backchain "with(r:rr,r=r);")
     (apply-macete-with-minor-premises
      int-defining-axiom_machine-arithmetic)
     simplify)))

  ))
```

Theorem B.9 (correctness-of-even%nn-odd%nn-lemma-2)
Theory: machine-arithmetic-extension
$\forall i : \mathbf{Z} \quad s.\ t. \quad 0 \leq i \wedge i \downarrow \text{int},$
conjunction

- $\Longleftrightarrow$
  - $\text{even\_nn}(i) = 1$
  - $\text{odd\_nn}(i) = 0$
- $\Longleftrightarrow$
  - $\text{odd\_nn}(i) = 1$
  - $\text{even\_nn}(i) = 0.$

```
(proof
 (

  (induction trivial-integer-inductor ("i"))
  (block
   (script-comment "‘induction’ at (0 0 0 0 0 0 0 0)")
   beta-reduce-repeatedly
   unfold-defined-constants
   (unfold-single-defined-constant-globally =_ma)
   (apply-macete-with-minor-premises rewrite-integer-constants))
  (block
```

```
     (script-comment
      "‘induction’ at (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)")
     (cut-with-single-formula "#(t,int)")
     (block
      (script-comment "‘cut-with-single-formula’ at (0)")
      (unfold-single-defined-constant-globally =_ma)
      (unfold-single-defined-constant-globally +_ma)
      (apply-macete-with-minor-premises rewrite-integer-constants)
      simplify)
     (block
      (script-comment "‘cut-with-single-formula’ at (1)")
      (incorporate-antecedent "with(r:rr,#(r,int));")
      (apply-macete-with-minor-premises
       int-defining-axiom_machine-arithmetic)
      simplify))

    ))
```

Theorem B.10 (correctness-of-even%nn)
Theory: machine-arithmetic-extension
$\forall i : \mathrm{int} \quad s.\ t. \quad 0 \leq i,$

$\iff$

- $\mathrm{even\_nn}(i) = 1$
- $\exists j : \mathrm{int} \quad i = 2 \cdot j.$

```
 (proof
   (


    direct-inference
    direct-inference
    (instantiate-theorem correctness-of-even%nn-odd%nn-lemma-1 ("i"))
    (contrapose "with(p:prop,not(p));")

    ))
```

Theorem B.11 (correctness-of-odd%nn)
Theory: machine-arithmetic-extension
$\forall i : \mathrm{int} \quad s.\ t. \quad 0 \leq i,$

$\iff$

- $\mathrm{odd\_nn}(i) = 1$
- $\exists j : \mathrm{int} \quad i = 2 \cdot j + 1.$

```
 (proof
   (
```

```
    direct-inference
    direct-inference
    (instantiate-theorem correctness-of-even%nn-odd%nn-lemma-1 ("i"))
    (contrapose "with(p:prop,not(p));")

    ))
```

Theorem B.12 (program-answer)
Theory: machine-arithmetic-extension
even_nn($\text{plus\_77}_{\text{ma}}$) = 0.

```
 (proof
  (

    (instantiate-theorem
     correctness-of-even%nn-odd%nn-lemma-2 ("plus%77_ma"))
    (block
     (script-comment "`instantiate-theorem' at (0 0 0)")
     (contrapose "with(p:prop,p);")
     (weaken (0))
     (apply-macete-with-minor-premises rewrite-integer-constants)
     simplify)
    (contrapose "with(p:prop,p);")
    (move-to-ancestor 2)
    (block
     (script-comment "`instantiate-theorem' at (0 1 0)")
     (backchain-backwards
      "odd%nn(plus%77_ma)=1 iff even%nn(plus%77_ma)=0;")
     (instantiate-theorem correctness-of-odd%nn ("plus%77_ma"))
     (contrapose "with(p:prop,forall(j:int,p));")
     (cut-with-single-formula "#(plus%77_ma,int)")
     (instantiate-existential ("38"))
     (block
      (script-comment "`instantiate-existential' at (0)")
      (apply-macete-with-minor-premises rewrite-integer-constants)
      simplify)
     (block
      (script-comment "`instantiate-existential' at (1)")
      (incorporate-antecedent "#(plus%77_ma,int);")
      (apply-macete-with-minor-premises rewrite-integer-constants)
      (apply-macete-with-minor-premises
       int-defining-axiom_machine-arithmetic)
      simplify))

    ))
```

Component theory: machine-arithmetic
Top level axioms:

machine-arithmetic-extension-axiom-0 $\mathrm{plus\_1}_{\mathrm{ma}} = 1.$

machine-arithmetic-extension-axiom-1 $\mathrm{zero}_{\mathrm{ma}} = 0.$

machine-arithmetic-extension-axiom-2 $\mathrm{minus\_1}_{\mathrm{ma}} = -1.$

machine-arithmetic-extension-axiom-3 $\mathrm{plus\_4}_{\mathrm{ma}} = 4.$

Figure 3: Components and axioms for machine-arithmetic-extension

C The File for Recursive Factorial Function

```
(include-files
 (files
  (imps /theories/machine-arithmetic/machine-arithmetic)))
```

Language C.1 (machine-arithmetic-language-extension)
Embedded language: *machine-arithmetic*
Constants: $\mathrm{plus\_1}_{\mathrm{ma}}$ *:* int
$\mathrm{zero}_{\mathrm{ma}}$ *:* int
$\mathrm{minus\_1}_{\mathrm{ma}}$ *:* int
$\mathrm{\_unspecified}_0$ *:* int
$\mathrm{\_unspecified}_1$ *:* int
$\mathrm{\_unspecified}_2$ *:* int
$\mathrm{plus\_4}_{\mathrm{ma}}$ *:* int

Theory C.2 (machine-arithmetic-extension)
Language: *machine-arithmetic-language-extension*
Component Theories and Axioms: *See Figure 3.*

Definition (Recursive) C.3 (fact)
Theory: machine-arithmetic-extension
$[\,\mathrm{fact} : \mathrm{int} \rightharpoonup \mathrm{int} \;\mapsto$
$\quad [\,n : \mathrm{int} \;\mapsto$

conditionally

- $\mathit{if}\ =_{\mathrm{ma}}(\mathrm{zero}_{\mathrm{ma}}, n)\ \mathit{then}\ \mathrm{plus\_1}_{\mathrm{ma}}$
- $\mathit{if}\ <_{\mathrm{ma}}(\mathrm{zero}_{\mathrm{ma}}, n)\ \mathit{then}$
  $[\,\mathrm{named\_by\_compiler} : \mathrm{int} \mapsto$
  $*_{\mathrm{ma}}(n, \mathrm{named\_by\_compiler})\,]$
  $(\mathrm{fact}(+_{\mathrm{ma}}(\mathrm{minus\_1}_{\mathrm{ma}}, n)))$
- $\mathit{otherwise}\ \mathrm{\_unspecified}_0\,]\,]$.

```
(view-expr "(apply-operator fact plus%4_ma)"
  (language machine-arithmetic-extension)
  (syntax sexp-syntax))

(def-compound-macete rewrite-integer-constants
  (series
   machine-arithmetic-extension-axiom-0
   machine-arithmetic-extension-axiom-1
   machine-arithmetic-extension-axiom-2
   machine-arithmetic-extension-axiom-3))
```

Theorem C.4 (fact-definedness-lemma)
Theory: machine-arithmetic-extension
$\forall n : \mathbf{Z}\quad s.\ t.\quad \mathrm{fact}(n)\downarrow,$
$n \downarrow \mathrm{int}.$

```
(proof
 (

  direct-inference
  (unfold-single-defined-constant-globally fact)
  simplify

  ))
```

Theorem C.5 (correctness-of-fact)
Theory: machine-arithmetic-extension
$\forall n : \mathbf{Z}\quad s.\ t.\quad 0 \leq n \land \mathrm{fact}(n)\downarrow,$
$\mathrm{fact}(n) = n!.$

```
(proof
 (

  (induction trivial-integer-inductor ("n"))
```

```
(block
 (script-comment "‘induction’ at (0 0 0 0 0 0 0 0)")
 beta-reduce-repeatedly
 direct-inference
 (unfold-single-defined-constant-globally fact)
 (unfold-single-defined-constant-globally =_ma)
 (apply-macete-with-minor-premises rewrite-integer-constants)
 simplify
 (unfold-single-defined-constant-globally factorial)
 (apply-macete-with-minor-premises tr%monoid-null-prod))
(move-to-ancestor 3)
(block
 (script-comment "‘induction’ at (0 0 0 0 0 0 0 1 0 0 0 0)")
 direct-inference
 (instantiate-theorem fact-definedness-lemma ("1+t"))
 (incorporate-antecedent "with(i:int,#(i));")
 (unfold-single-defined-constant-globally fact)
 (unfold-single-defined-constant-globally =_ma)
 (unfold-single-defined-constant-globally <_ma)
 (unfold-single-defined-constant-globally +_ma)
 (apply-macete-with-minor-premises rewrite-integer-constants)
 simplify
 direct-and-antecedent-inference-strategy
 (apply-macete-with-minor-premises unfold-defined-expression%*_ma)
 (backchain "with(p:prop,p implies p);")
 direct-inference
 (apply-macete-locally factorial-out (0) "(1+t)!")
 simplify)

))
```

Theorem C.6 (program-answer)
Theory: machine-arithmetic-extension
implication
- $\mathrm{fact}(\mathrm{plus\_4}_{\mathrm{ma}}) \downarrow$
- $\mathrm{fact}(\mathrm{plus\_4}_{\mathrm{ma}}) = 4!$.

```
(proof
  (

   (apply-macete-with-minor-premises rewrite-integer-constants)
   direct-inference
   (apply-macete-with-minor-premises correctness-of-fact)

   ))
```

Component theory: machine-arithmetic
Top level axioms:

machine-arithmetic-extension-axiom-0 $\mathrm{zero}_{\mathrm{ma}} = 0.$

machine-arithmetic-extension-axiom-1 $\mathrm{minus\_1}_{\mathrm{ma}} = -1.$

machine-arithmetic-extension-axiom-2 $\mathrm{plus\_4}_{\mathrm{ma}} = 4.$

machine-arithmetic-extension-axiom-3 $\mathrm{plus\_1}_{\mathrm{ma}} = 1.$

Figure 4: Components and axioms for machine-arithmetic-extension

D The File for Iterative Factorial Function

```
(include-files
 (files
  (imps /theories/machine-arithmetic/machine-arithmetic)))
```

Language D.1 (machine-arithmetic-language-extension)
Embedded language: *machine-arithmetic*
Constants: $\mathrm{zero}_{\mathrm{ma}}$ *:* int
$\mathrm{minus\_1}_{\mathrm{ma}}$ *:* int
$\mathrm{\_unspecified}_0$ *:* int
$\mathrm{\_unspecified}_1$ *:* int
$\mathrm{\_unspecified}_2$ *:* int
$\mathrm{\_unspecified}_3$ *:* int
$\mathrm{plus\_4}_{\mathrm{ma}}$ *:* int
$\mathrm{plus\_1}_{\mathrm{ma}}$ *:* int

Theory D.2 (machine-arithmetic-extension)
Language: *machine-arithmetic-language-extension*
Component Theories and Axioms: *See Figure 4.*

Definition (Recursive) D.3 (fact%loop)
Theory: machine-arithmetic-extension

$[\,\mathrm{fact\_loop} : \mathrm{int} \times \mathrm{int} \rightharpoonup \mathrm{int} \ \mapsto$
  $[\,n, a : \mathrm{int} \ \mapsto$
    conditionally, if $<_{\mathrm{ma}} (\mathrm{zero}_{\mathrm{ma}}, n)$
    - *then* $\mathrm{fact\_loop}(+_{\mathrm{ma}}(\mathrm{minus\_1}_{\mathrm{ma}}, n), *_{\mathrm{ma}}(n, a))$
    - *else* $a\,]\,]$.

```
(view-expr "(apply-operator fact%loop plus%4_ma plus%1_ma)"
  (language machine-arithmetic-extension)
  (syntax sexp-syntax))
```

```
(def-compound-macete rewrite-integer-constants
  (series
   machine-arithmetic-extension-axiom-0
   machine-arithmetic-extension-axiom-1
   machine-arithmetic-extension-axiom-2
   machine-arithmetic-extension-axiom-3))
```

Theorem D.4 (fact%loop-definedness-lemma)
Theory: machine-arithmetic-extension
$\forall n, a : \mathbf{Z} \quad s.\ t. \quad \mathrm{fact\_loop}(n, a) \downarrow,$
  conjunction
  - $n \downarrow \mathrm{int}$
  - $a \downarrow \mathrm{int}.$

```
(proof
  (

   direct-inference
   (unfold-single-defined-constant-globally fact%loop)
   simplify

   ))
```

Theorem D.5 (correctness-of-fact%loop-lemma)
Theory: machine-arithmetic-extension
$\forall n : \mathbf{Z} \quad s.\ t. \quad 0 \leq n,$
  $\forall a : \mathbf{Z} \quad s.\ t. \quad \mathrm{fact\_loop}(n, a) \downarrow,$
  $\mathrm{fact\_loop}(n, a) = n! \cdot a.$

```
(proof
  (
```

```
(induction trivial-integer-inductor ("n"))
(block
 (script-comment "`induction' at (0 0)")
 beta-reduce-repeatedly
 direct-and-antecedent-inference-strategy
 (unfold-single-defined-constant-globally fact%loop)
 unfold-defined-constants
 (apply-macete-with-minor-premises rewrite-integer-constants)
 (apply-macete-with-minor-premises tr%monoid-null-prod)
 simplify)
(move-to-ancestor 5)
(block
 (script-comment "`induction' at (0 1 0 0 0)")
 direct-and-antecedent-inference-strategy
 (instantiate-theorem fact%loop-definedness-lemma ("1+t" "a"))
 (cut-with-single-formula "#(fact%loop(t,(1+t) *_ma a))")
 (block
  (script-comment "`cut-with-single-formula' at (0)")
  (unfold-single-defined-constant-globally fact%loop)
  (unfold-single-defined-constant-globally <_ma)
  (unfold-single-defined-constant-globally +_ma)
  (apply-macete-with-minor-premises rewrite-integer-constants)
  simplify
  (backchain "with(p:prop,forall(a:zz,p));")
  direct-inference
  (instantiate-theorem
   fact%loop-definedness-lemma ("t" "(1+t) *_ma a"))
  (apply-macete-with-minor-premises
   unfold-defined-expression%*_ma)
  (apply-macete-locally-with-minor-premises
   factorial-out (0) "(1+t)!")
  simplify)
 (block
  (script-comment "`cut-with-single-formula' at (1)")
  (incorporate-antecedent "with(i:int,#(i));")
  (unfold-single-defined-constant (0) fact%loop)
  (unfold-single-defined-constant-globally <_ma)
  (unfold-single-defined-constant-globally +_ma)
  (apply-macete-with-minor-premises rewrite-integer-constants)
  simplify))

))
```

Theorem D.6 (correctness-of-fact%loop)

Theory: machine-arithmetic-extension

$\forall n, a : \mathbf{Z} \quad s.\ t. \quad 0 \leq n \wedge \mathrm{fact\_loop}(n, a) \downarrow,$
$\quad \mathrm{fact\_loop}(n, a) = n! \cdot a.$

```
(proof
  (

   direct-and-antecedent-inference-strategy
   (apply-macete-with-minor-premises
    correctness-of-fact%loop-lemma)

   ))
```

Theorem D.7 (program-answer)
Theory: machine-arithmetic-extension
implication

- $\text{fact\_loop}(\text{plus\_4}_{\text{ma}}, \text{plus\_1}_{\text{ma}}) \downarrow$
- $\text{fact\_loop}(\text{plus\_4}_{\text{ma}}, \text{plus\_1}_{\text{ma}}) = 4!$.

```
(proof
  (

   (apply-macete-with-minor-premises rewrite-integer-constants)
   direct-inference
   (apply-macete-with-minor-premises correctness-of-fact%loop)
   simplify

   ))
```

E The File for Fibonacci Function

```
(include-files
 (files
  (imps /theories/machine-arithmetic/machine-arithmetic)))
```

Language E.1 (machine-arithmetic-language-extension)
Embedded language: *machine-arithmetic*
Constants: zero_{ma} *:* int
$\text{minus\_1}_{\text{ma}}$ *:* int
_unspecified_0 *:* int
_unspecified_1 *:* int
_unspecified_2 *:* int
_unspecified_3 *:* int
$\text{plus\_1}_{\text{ma}}$ *:* int

Component theory: machine-arithmetic
Top level axioms:

machine-arithmetic-extension-axiom-0 $\mathrm{zero}_{\mathrm{ma}} = 0.$

machine-arithmetic-extension-axiom-1 $\mathrm{minus\_1}_{\mathrm{ma}} = -1.$

machine-arithmetic-extension-axiom-2 $\mathrm{plus\_1}_{\mathrm{ma}} = 1.$

Figure 5: Components and axioms for machine-arithmetic-extension

Theory E.2 (machine-arithmetic-extension)
Language: *machine-arithmetic-language-extension*
Component Theories and Axioms: *See Figure 5.*

The following 2 definitions are mutually recursive.

Definition (Recursive) E.3 (fib%2)
Theory: machine-arithmetic-extension
$[\,\mathrm{fib\_2}, \mathrm{fib\_1} : \mathrm{int} \rightharpoonup \mathrm{int} \mapsto$
$[\,\mathrm{n1} : \mathrm{int} \mapsto$
conditionally, if $=_{\mathrm{ma}} (\mathrm{zero}_{\mathrm{ma}}, \mathrm{n1})$
- *then* $\mathrm{zero}_{\mathrm{ma}}$
- *else* $\mathrm{fib\_1}(+_{\mathrm{ma}}(\mathrm{minus\_1}_{\mathrm{ma}}, \mathrm{n1}))\,]\,].$

Definition (Recursive) E.4 (fib%1)
Theory: machine-arithmetic-extension
$[\,\mathrm{fib\_2}, \mathrm{fib\_1} : \mathrm{int} \rightharpoonup \mathrm{int} \mapsto$
$[\,n : \mathrm{int} \mapsto$
conditionally, if $=_{\mathrm{ma}} (\mathrm{zero}_{\mathrm{ma}}, n)$
- *then* $\mathrm{plus\_1}_{\mathrm{ma}}$
- *else* $[\,\mathrm{named\_by\_compiler}, \mathrm{named\_by\_compiler1} : \mathrm{int} \mapsto$
  $+_{\mathrm{ma}} (\mathrm{named\_by\_compiler}, \mathrm{named\_by\_compiler1})\,]$
  $(\mathrm{fib\_1}(+_{\mathrm{ma}}(\mathrm{minus\_1}_{\mathrm{ma}}, n)), \mathrm{fib\_2}(+_{\mathrm{ma}}(\mathrm{minus\_1}_{\mathrm{ma}}, n)))\,]\,].$

```
(view-expr "(apply-operator fib%1 plus%1_ma)"
  (language machine-arithmetic-extension)
  (syntax sexp-syntax))
```

Definition (Recursive) E.5 (fib)
Theory: h-o-real-arithmetic
$[\, f : \mathbf{Z} \rightharpoonup \mathbf{R} \mapsto$
$\quad [\, n : \mathbf{Z} \mapsto$
$\qquad$ *conditionally*
- *if* $n = 0$ *then* 1
- *if* $n = 1$ *then* 1
- *otherwise* $f(n-1) + f(n-2)\,]\,]$.

Theorem E.6 (uniqueness-for-fibonacci)
Theory: h-o-real-arithmetic
$\forall f : \mathbf{Z} \rightharpoonup \mathbf{R}, n : \mathbf{Z}$ implication
- conjunction
  - ○ $\forall x : \mathbf{Z} \quad s.\ t. \quad 2 \leq x \wedge x \leq n,$
$f(x) = f(x-1) + f(x-2)$
  - ○ $f(0) = 1$
  - ○ $f(1) = 1$
- $\forall x : \mathbf{Z} \quad s.\ t. \quad 0 \leq x \wedge x \leq n,$
$f(x) = \mathrm{fib}(x)$.

```
(proof
 (


  direct-inference
  direct-inference
  (antecedent-inference "with(p:prop,p);")
  (induction complete-inductor ("x"))
  (case-split ("m=0"))
  simplify
  (block
    (script-comment "‘case-split’ at (2)")
    (case-split ("m=1"))
    simplify
    (block
      (script-comment "‘case-split’ at (2)")
      (cut-with-single-formula "2<=m")
      (move-to-sibling 1)
      simplify
      (block
        (script-comment "‘cut-with-single-formula’ at (0)")
        simplify
        (backchain
         "with(r:rr,p:prop,forall(x:zz,p and p implies r=r));")
        (backchain
```

```
          "with(p:prop,forall(k:zz,p and p implies (p implies p)));")
         (backchain
          "with(p:prop,forall(k:zz,p and p implies (p implies p)));")
         simplify
         direct-inference
         (block
           (script-comment "`direct-inference' at (0)")
           (instantiate-universal-antecedent
            "with(p:prop,forall(k:zz,p and p implies (p implies p)));"
            ("[-1]+m"))
           (simplify-antecedent "with(r:rr,not(0<=r));")
           (simplify-antecedent "with(m:zz,r:rr,not(r<m));")
           (simplify-antecedent "with(n:zz,r:rr,not(r<=n));"))
         (block
           (script-comment "`direct-inference' at (1)")
           (instantiate-universal-antecedent
            "with(p:prop,forall(k:zz,p and p implies (p implies p)));"
            ("[-2]+m"))
           (simplify-antecedent "with(r:rr,not(0<=r));")
           (simplify-antecedent "with(m:zz,r:rr,not(r<m));")
           (simplify-antecedent "with(n:zz,r:rr,not(r<=n));")))))

   ))
```

```
(def-compound-macete apply-machine-axioms
  (repeat
   machine-arithmetic-extension-axiom-0
   machine-arithmetic-extension-axiom-1
   machine-arithmetic-extension-axiom-2))

(def-script unfold-machine-constants 0
  (
   (while
    (progresses?
     (block
       (apply-macete-with-minor-premises apply-machine-axioms)
       (unfold-single-defined-constant-globally =_ma)
       (unfold-single-defined-constant-globally +_ma)
       (unfold-single-defined-constant-globally *_ma)))
    (skip))))
```

Theorem E.7 (fib%1-recursive-definedness-lemma)
Theory: machine-arithmetic-extension

$\forall x:\mathbf{Z}$ implication

- conjunction
  - ○ $1 \leq x$
  - ○ $x \leq$ maxint
  - ○ fib_1$(x)\downarrow$
- conjunction
  - ○ fib_1$(x-1)\downarrow$ int
  - ○ fib_2$(x-1)\downarrow$ int.

```
(proof
  (

   (unfold-single-defined-constant (0) fib%1)
   unfold-machine-constants
   simplify

   ))
```

Theorem E.8 (subtraction-lemma)
Theory: machine-arithmetic-extension
$\forall x:\text{int}, y:\mathbf{Z}\quad s.\ t.\quad 0 \leq y \wedge y \leq x,$
$x - y \downarrow \text{int}.$

```
(proof
  (

   direct-and-antecedent-inference-strategy
   (cut-with-single-formula "#(x,int)")
   (incorporate-antecedent "with(x:int,#(x,int));")
   (apply-macete-with-minor-premises
    int-defining-axiom_machine-arithmetic)
   beta-reduce-repeatedly
   (apply-macete-with-minor-premises minint-is-negative-maxint)
   simplify

   ))
```

Theorem E.9 (minus-1-lemma)
Theory: machine-arithmetic-extension
$\forall x:\text{int}\quad s.\ t.\quad 1 \leq x,$
$+_{\text{ma}}(\text{minus\_1}_{\text{ma}}, x) = x - 1.$

```
(proof
  (
```

```
  unfold-machine-constants
  direct-and-antecedent-inference-strategy
  (cut-with-single-formula "#([-1]+x,int)")
  simplify
  (block
    (script-comment "‘cut-with-single-formula’ at (1)")
    (cut-with-single-formula "#(x-1,int)")
    (simplify-antecedent "with(r:rr,#(r,int));")
    (block
      (script-comment "‘cut-with-single-formula’ at (1)")
      (apply-macete-with-minor-premises subtraction-lemma)
      simplify))

  ))
```

Theorem E.10 (fib%1-recursive-condition)

Theory: machine-arithmetic-extension

$\forall x : \mathbf{Z}$ implication

- conjunction
  - $2 \leq x$
  - $x \leq \text{maxint}$
  - $\text{fib\_1}(x) \downarrow$
- $\text{fib\_1}(x) = \text{fib\_1}(x-1) + \text{fib\_1}(x-2)$.

```
(proof
 (

  (unfold-single-defined-constant (0) fib%1)
  (apply-macete-with-minor-premises minus-1-lemma)
  unfold-machine-constants
  simplify
  direct-and-antecedent-inference-strategy
  (unfold-single-defined-constant (0) fib%1)
  (apply-macete-with-minor-premises minus-1-lemma)
  unfold-machine-constants
  simplify
  direct-and-antecedent-inference-strategy
  (block
   (script-comment
    "‘direct-and-antecedent-inference-strategy’ at (0)")
   (contrapose "with(r:rr,#(r));")
   simplify)
  (block
   (script-comment
    "‘direct-and-antecedent-inference-strategy’ at (1)")
```

```
 (unfold-single-defined-constant (0) fib%2)
 (apply-macete-with-minor-premises minus-1-lemma)
 unfold-machine-constants
 simplify
 (cut-with-single-formula
  "#(fib%1((x_$0-1)-1),int) and  #(fib%2((x_$0-1)-1),int)")
 (block
  (script-comment "`cut-with-single-formula' at (0)")
  (antecedent-inference "with(p:prop,p and p);")
  (contrapose "with(r:rr,#(fib%1(r),int));")
  simplify)
 (block
  (script-comment "`cut-with-single-formula' at (1)")
  (apply-macete-with-minor-premises
   fib%1-recursive-definedness-lemma)
  simplify))

))
```

Theorem E.11 (hereditary-definedness-of-fib%1)

Theory: machine-arithmetic-extension

$\forall x, y : \mathbf{Z}$ implication

- conjunction
  - $0 \leq x$
  - $0 \leq y$
  - $y \leq x$
  - fib_1$(x) \downarrow$
- fib_1$(y) \downarrow$.

```
(proof
  (

   (cut-with-single-formula
    "forall(x,y:zz,
       0<=x and 0<=y and y<=x and #(fib%1(x)) implies #(fib%1(x-y)))")
   (block
    (script-comment "`cut-with-single-formula' at (0)")
    direct-and-antecedent-inference-strategy
    (instantiate-universal-antecedent "with(p:prop,forall(x,y:zz,p));"
                                      ("x" "x-y"))
    (simplify-antecedent "with(p:prop,not(p));")
    (simplify-antecedent "with(p:prop,not(p));")
    (simplify-antecedent "with(r:rr,x:zz,#(fib%1(x-r)));"))
   (block
    (script-comment "`cut-with-single-formula' at (1)")
    (induction trivial-integer-inductor ("y"))
```

```
simplify
(move-to-ancestor 5)
(block
 (script-comment "‘induction’ at (0 0 0 0 0 0 0 1)")
 beta-reduce-repeatedly
 direct-and-antecedent-inference-strategy
 (simplify-antecedent "with(p:prop,not(p));")
 (block
  (script-comment
   "‘direct-and-antecedent-inference-strategy’ at (0 0 0 0 1)")
  (case-split ("t+1=x"))
  (block
   (script-comment "‘case-split’ at (1)")
   simplify
   (unfold-single-defined-constant (0) fib%1)
   unfold-machine-constants)
  (block
   (script-comment "‘case-split’ at (2)")
   (cut-with-single-formula
    "fib%1(x-t)=fib%1((x-t)-1) + fib%1((x-t)-2)")
   (block
    (script-comment "‘cut-with-single-formula’ at (0)")
    simplify
    (simplify-antecedent "with(r:rr,i:int,i=r);"))
   (block
    (script-comment "‘cut-with-single-formula’ at (1)")
    (instantiate-theorem fib%1-recursive-condition
                         ("x-t"))
    (block
     (script-comment "‘instantiate-theorem’ at (0 0 0)")
     (cut-with-single-formula "x<=maxint")
     (simplify-antecedent "with(r:rr,not(2<=r));")
     (block
      (script-comment "‘cut-with-single-formula’ at (1)")
      (cut-with-single-formula "#(x,int)")
      (incorporate-antecedent "with(x:zz,#(x,int));")
      (apply-macete-with-minor-premises
       int-defining-axiom_machine-arithmetic)
      simplify))
    (block
     (script-comment "‘instantiate-theorem’ at (0 0 1)")
     (cut-with-single-formula "#(x,int)")
     (incorporate-antecedent "with(x:zz,#(x,int));")
     (apply-macete-with-minor-premises
      int-defining-axiom_machine-arithmetic)
     beta-reduce-repeatedly
     direct-and-antecedent-inference-strategy
```

```
        (simplify-antecedent "with(r:rr,not(r<=maxint));")))))))

   ))
```

Theorem E.12 (fib%1-if-defined-is-fib)
Theory: machine-arithmetic-extension
$\forall x : \mathbf{Z} \quad s.\ t. \quad 0 \leq x \wedge \text{fib\_1}(x) \downarrow,$
$\text{fib\_1}(x) = \text{fib}(x).$

```
(proof
  (

   direct-and-antecedent-inference-strategy
   (instantiate-theorem uniqueness-for-fibonacci ("fib%1" "x"))
   (block
     (script-comment "‘instantiate-theorem’ at (0 0 0 0 0)")
     (contrapose "with(p:prop,not(p));")
     (instantiate-theorem fib%1-recursive-condition ("x_$0"))
     (block
       (script-comment "‘instantiate-theorem’ at (0 0 1)")
       (cut-with-single-formula "#(x_$0,int)")
       (block
         (script-comment "‘cut-with-single-formula’ at (0)")
         (incorporate-antecedent "with(x_$0:zz,#(x_$0,int));")
         (apply-macete-with-minor-premises
          int-defining-axiom_machine-arithmetic)
         beta-reduce-repeatedly
         direct-and-antecedent-inference-strategy)
       (block
         (script-comment "‘cut-with-single-formula’ at (1)")
         (cut-with-single-formula "#(fib%1(x_$0))")
         (apply-macete-with-minor-premises
          hereditary-definedness-of-fib%1)
         (instantiate-existential ("x"))
         simplify))
     (block
       (script-comment "‘instantiate-theorem’ at (0 0 2)")
       (contrapose "with(i:int,not(#(i)));")
       (apply-macete-with-minor-premises
        hereditary-definedness-of-fib%1)
       (instantiate-existential ("x"))
       simplify))
   (block
     (script-comment "‘instantiate-theorem’ at (0 0 1)")
     (contrapose "with(p:prop,not(p));")
     (unfold-single-defined-constant (0) fib%1)
     unfold-machine-constants)
```

```
(block
  (script-comment "‘instantiate-theorem’ at (0 0 2)")
  (contrapose "with(p:prop,not(p));")
  (unfold-single-defined-constant (0) fib%1)
  (apply-macete-with-minor-premises minus-1-lemma)
  unfold-machine-constants
  simplify
  (unfold-single-defined-constant (0) fib%2)
  unfold-machine-constants
  simplify)
simplify

))
```

F The File for Greatest Common Denominator

```
(include-files
 (files (imps /theories/machine-arithmetic/gcd)))
```

Language F.1 (machine-arithmetic-language-extension)
Embedded language: *machine-arithmetic*
Constants: $\mathrm{zero}_{\mathrm{ma}}$ *:* int
$\_\mathrm{unspecified}_0$ *:* int
$\_\mathrm{unspecified}_1$ *:* int
$\_\mathrm{unspecified}_2$ *:* int
$\mathrm{plus\_6}_{\mathrm{ma}}$ *:* int
$\mathrm{plus\_7}_{\mathrm{ma}}$ *:* int

Theory F.2 (machine-arithmetic-extension)
Language: *machine-arithmetic-language-extension*
Component Theories and Axioms: *See Figure 6.*

Definition (Recursive) F.3 (gcd_scm)
Theory: machine-arithmetic-extension
$[\,\mathrm{gcd}_{\mathrm{scm}} : \mathrm{int} \times \mathrm{int} \rightharpoonup \mathrm{int} \mapsto$
$\quad [\,u, v : \mathrm{int} \mapsto$
$\qquad$ *conditionally, if*
$\qquad$ *if* $\leq_{\mathrm{ma}} (\mathrm{zero}_{\mathrm{ma}}, v)$ *then* $\leq_{\mathrm{ma}} (\mathrm{zero}_{\mathrm{ma}}, u)$ *else* falsehood
- *then* *if* $=_{\mathrm{ma}} (\mathrm{zero}_{\mathrm{ma}}, u)$

Component theory: machine-arithmetic
Top level axioms:

machine-arithmetic-extension-axiom-0 $\mathrm{zero}_{\mathrm{ma}} = 0.$

machine-arithmetic-extension-axiom-1 $\mathrm{plus\_6}_{\mathrm{ma}} = 6.$

machine-arithmetic-extension-axiom-2 $\mathrm{plus\_7}_{\mathrm{ma}} = 7.$

Figure 6: Components and axioms for machine-arithmetic-extension

  - ○ *then* v
  - ○ *else* *if* $=_{\mathrm{ma}} (\mathrm{zero}_{\mathrm{ma}}, v)$ *then* u *else* $\mathrm{gcd}_{\mathrm{scm}}(v, \mathrm{mod}_{\mathrm{ma}}(u, v))$
- *else* $\_\mathrm{unspecified}_0\,]\,]$.

```
(view-expr "(apply-operator gcd_scm plus%6_ma plus%7_ma)"
  (language machine-arithmetic-extension)
  (syntax sexp-syntax))
```

Theorem F.4 (gcd_scm-is-gcd)
Theory: machine-arithmetic-extension
$\forall a, b : \mathrm{int} \quad s.\ t. \quad 0 \leq a \wedge 0 \leq b,$
$\quad \mathrm{gcd}_{\mathrm{scm}}(a, b) = \mathrm{gcd}(a, b).$

```
 (proof
  (


   (cut-with-single-formula
    "forall(b:zz,
       0<=b and b<=maxint
        implies
       forall(a:zz, 0<=a and a<=maxint implies gcd_scm(a,b)=gcd(a,b)))")
   (block
    (script-comment "‘cut-with-single-formula’ at (0)")
    direct-and-antecedent-inference-strategy
    (backchain "with(p:prop,forall(b:zz,p));")
    direct-and-antecedent-inference-strategy
    (block
     (script-comment
      "‘direct-and-antecedent-inference-strategy’ at (0 1)")
     (cut-with-single-formula "#(b,int)")
     (incorporate-antecedent "with(b:int,#(b,int));")
```

```
  (apply-macete-with-minor-premises
   int-defining-axiom_machine-arithmetic)
  beta-reduce-repeatedly
  direct-and-antecedent-inference-strategy)
 (block
  (script-comment
   "‘direct-and-antecedent-inference-strategy’ at (1 1 0)")
  (cut-with-single-formula "#(a,int)")
  (incorporate-antecedent "with(a:int,#(a,int));")
  (apply-macete-with-minor-premises
   int-defining-axiom_machine-arithmetic)
  beta-reduce-repeatedly
  direct-and-antecedent-inference-strategy))
(block
 (script-comment "‘cut-with-single-formula’ at (1)")
 (induction complete-inductor ("b"))
 (apply-macete-with-minor-premises
  machine-arithmetic-extension-axiom-0)
 (case-split ("a=0"))
 (block
  (script-comment "‘case-split’ at (1)")
  simplify
  (apply-macete-with-minor-premises symmetry-of-gcd)
  (apply-macete-with-minor-premises gcd-for-zero)
  (unfold-single-defined-constant-globally <=_ma)
  (unfold-single-defined-constant-globally =_ma)
  simplify)
 (block
  (script-comment "‘case-split’ at (2)")
  (unfold-single-defined-constant-globally =_ma)
  (unfold-single-defined-constant-globally <=_ma)
  simplify
  (case-split ("m=0"))
  (block
   (script-comment "‘case-split’ at (1)")
   simplify
   (apply-macete-with-minor-premises gcd-for-zero))
  (block
   (script-comment "‘case-split’ at (2)")
   simplify
   beta-reduce-with-minor-premises
   (move-to-sibling 1)
   (block
    (script-comment "‘beta-reduce-with-minor-premises’ at (1)")
    (unfold-single-defined-constant (0) mod_ma)
    (cut-with-single-formula "#(a mod m ,zz)")
    (apply-macete-with-minor-premises mod-of-integer-is-integer))
```

```
(block
 (script-comment "‘beta-reduce-with-minor-premises’ at (0)")
 (unfold-single-defined-constant-globally mod_ma)
 (instantiate-theorem division-with-remainder
                      ("m" "a"))
 simplify
 (case-split ("a mod m = 0"))
 (block
  (script-comment "‘case-split’ at (1)")
  simplify
  (contrapose "with(r:rr,r=0);")
  (apply-macete-with-minor-premises mod-characterization)
  simplify
  (contrapose "with(a,m:zz,not(m=gcd(a,m)));")
  (apply-macete-with-minor-premises gcd-of-multiple))
 (block
  (script-comment "‘case-split’ at (2)")
  simplify
  (backchain "with(p:prop,forall(k:zz,p));")
  (move-to-sibling 1)
  (apply-macete-with-minor-premises mod-of-integer-is-integer)
  (block
   (script-comment "‘backchain’ at (0)")
   (instantiate-theorem division-with-remainder
                        ("a mod m" "m"))
   direct-and-antecedent-inference-strategy
   simplify
   simplify
   (block
    (script-comment
     "‘direct-and-antecedent-inference-strategy’ at (1 1 0 1 0)")
    (apply-macete-with-minor-premises symmetry-of-gcd)
    simplify)
   (block
    (script-comment
     "‘direct-and-antecedent-inference-strategy’ at (1 1 1 0 0)")
    (apply-macete-with-minor-premises symmetry-of-gcd)
    (block
     (script-comment
      "‘apply-macete-with-minor-premises’ at (0)")
     (apply-macete-with-minor-premises rev%invariance-of-gcd)
     (apply-macete-with-minor-premises symmetry-of-gcd)
     (apply-macete-with-minor-premises rev%invariance-of-gcd)
     simplify
     (unfold-single-defined-constant (0) gcd)
     (apply-macete-with-minor-premises
      definedness-of-generator)
```

```
         (apply-macete-with-minor-premises
          integer-combinations-form-an-ideal))
        (apply-macete-with-minor-premises
         mod-of-integer-is-integer)))))))

))
```

References

[1] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

[2] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[3] W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, 1993. Available at `http://imps.mcmaster.ca/`.

[4] J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A guide to VLISP, a verified programming language implementation. M 92B91, The MITRE Corporation, September 1992.

[5] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[6] R. A. Kelsey. PreScheme: A Scheme dialect for systems programming. 1992.

[7] D. P. Oliva, J. D. Ramsdell, and M. Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1/2):111–182, 1995.

[8] M. Payne, C. Schaffert, and B. Wichmann. Proposal for a language compatible arithmetic standard. *SIGNUM Newsletter*, 25:2–43, January 1990.

[9] J. D. Ramsdell. The revised VLISP PreScheme front end. M 93B95, The MITRE Corporation, August 1993.