Formalización de la aritmética de TLA^+ en el asistente de pruebas Isabelle

Tesina de grado presentada
por

Hernán P. Vanzetto
V-1320/0

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
República Argentina

Director: Stephan Merz
INRIA Nancy, Francia

18 de Junio de 2010

Resumen

TLA$^+$ es un lenguaje para especificar sistemas distribuidos y concurrentes. Está basado en una lógica clásica de primer orden no-tipada y en una variante de la teoría de conjuntos estándar ZF, más una pequeña parte de lógica temporal. Una versión extendida del lenguaje, llamada TLA^{+2}, permite además escribir pruebas estructuradas jerárquicamente para verificar propiedades de las especificaciones.

El Administrador de Pruebas TLAPM transforma las pruebas escritas en TLA^{+2} en una colección de obligaciones de prueba que son enviadas a uno o más demostradores secundarios para que sean verificadas. Estos producen trazas o scripts de las pruebas verificadas que luego deben certificarse en un entorno lógico como el asistente de pruebas genérico Isabelle. De esta forma, el núcleo del entorno lógico es el único componente confiable del sistema de pruebas de TLA$^+$.

El lenguaje TLA$^+$ está siendo formalizado en Isabelle como una nueva lógica-objeto llamada Isabelle/TLA$^+$. Hasta el momento incluye un subconjunto de la lógica de primer orden, teoría de conjuntos, funciones, puntos fijos y la construcción de números naturales, y se instanciaron los principales métodos de prueba semi-automáticos ya existentes en Isabelle.

El objetivo de este trabajo es extender Isabelle/TLA$^+$ para dar soporte a la aritmética estándar de TLA$^+$ sobre los números naturales y enteros. Esto implica definir axiomáticamente los operadores aritméticos y probar sus propiedades para aumentar el poder de razonamiento de los métodos de prueba automáticos, lo que permitirá al TLAPM certificar las pruebas de especificaciones que utilizan aritmética.

Encoding TLA$^+$ arithmetic in the Isabelle proof assistant

Abstract

TLA$^+$ is a language for specifying and verifying concurrent and distributed systems. It is based in an untyped classical first-order logic and in a variant of the standard ZF set theory, plus a small part of temporal logic. An extended version of the language, called TLA^{+2}, allows writing proofs in a hierarchical style, enabling the inferential verification of TLA$^+$ specifications.

The TLA Proof Manager transforms the TLA^{+2} proofs in a collection of proof obligations that are sent to one or more back-end provers to be verified. For each verified proof, they produce a proof script that must be certified in a logical framework like the generic proof assistant Isabelle. In this way, the kernel of the logical framework is the only trustable component of the TLA$^+$ Proof System.

The core of TLA$^+$ has been encoded in Isabelle as a new object-logic called Isabelle/TLA$^+$. It currently includes a subset of first-order logic, set theory, functions, fixed points, and the construction of natural numbers. The main existing semi-automatic proof methods of Isabelle have been instantiated.

The subject of this work is to extend Isabelle/TLA$^+$ to provide support for the standard arithmetic of TLA$^+$ over natural and integer numbers. This implies to axiomatically define the arithmetic operators and to prove their properties to expand the reasoning of the automatic proof methods, allowing the Proof Manager to certify proofs of specifications that use arithmetic.

Índice general

Índice de figuras

Capítulo 1

Introducción

A medida que los sistemas de hardware y software se vuelven más complejos es más probable que se generen fallas durante su desarrollo, y a su vez, que estas fallas sean más difíciles de detectar por los métodos de testing tradicionales. Hay varios ejemplos de sistemas críticos en los que fallas o errores debidos a un mal diseño han causado pérdidas humanas y materiales [Pro96, Cor94, For03]. Los métodos de verificación formal usan técnicas matemáticas para tratar este tipo de problemas.

Comúnmente, se usa el término *verificación* para llamar al proceso que busca asegurar que el diseño de un sistema satisface las propiedades que expresan el correcto funcionamiento del mismo. El aspecto formal y preciso se logra al describir el sistema y sus propiedades en un lenguaje matemático. En general, se distinguen dos tipos de propiedades: las propiedades de *vivacidad*, que expresan que el sistema hace lo que se supone que debe hacer, y las propiedades de *seguridad*, que expresan que el sistema no hace nada que no se desee.

Durante la verificación de un sistema debemos hacernos la pregunta "¿estamos construyendo el producto correctamente?", y durante la validación, "¿estamos construyendo el producto correcto?". Hacer testing es validar un sistema. Por lo tanto, se dice que verificación y validación son aspectos complementarios.

A su vez, la verificación puede encararse de dos formas opuestas, llamadas verificación deductiva y verificación semántica (aunque en los últimos años también se considera dentro del área de verificación al *testing formal*).

En la *verificación deductiva* se quiere probar que de la especificación $\mathcal{F}$ se deduce sintácticamente la propiedad ϕ ($\mathcal{F} \vdash \phi$) aplicando reglas de inferencia. Se usan propiedades del diseño para derivar pruebas de que el sistema exhibe todas y sólo las propiedades requeridas. En la mecanización de esta técnica se usan asistentes de pruebas o demostradores de teoremas, como Isabelle, Coq o HOL. Es por ello que esta área es más conocida como *theorem proving*.

En la *verificación semántica* se quiere verificar que el modelo $\mathcal{K}$ satisface semánticamente la propiedad ϕ ($\mathcal{K} \models \phi$). Este método algorítmico de verificación es más conocido con el nombre de *model-checking*. En esta técnica se examina exhaustivamente todos los estados posibles en que puede estar el sistema, y se somete cada estado (o secuencias de estados) a todas las posibles combinaciones de estímulos internos y externos, para luego chequear que sólo las propiedades deseadas se cumplen (es decir, que el sistema nunca se encuentre en un estado no deseado). A diferencia de la verificación por pruebas, este método está mucho más automatizado, aunque se debe reducir la especificación a un modelo finito para no producir una explosión de los estados de búsqueda. Es también un buen complemento de aquel para detectar errores al comienzo de la verificación.

Verificación de especificaciones en TLA^{+2}

El proyecto en el cual se desarrolló este trabajo [tlaa], tiene por objetivo de fondo verificar especificaciones escritas en el lenguaje TLA$^+$ usando demostradores de teoremas, uno de los cuales es Isabelle. TLA$^+$ es un lenguaje para especificar y verificar sistemas distribuidos y concurrentes [tlab]. Está basado en una variante de la teoría de conjuntos de Zermelo-Fränkel (ZF) para describir las estructuras de datos manipuladas por los algoritmos a verificar, y en la Lógica Temporal de Acciones (Temporal Logic of Actions, TLA) para describir el comportamiento dinámico de los mismos.

La especificación de un sistema en TLA$^+$ consiste generalmente en una gran parte de matemática ordinaria combinada con una pequeña parte de lógica temporal. Las propiedades de sistemas que se quieren verificar se escriben en el lenguaje de la herramienta de verificación que se esté usando, que por lo general es distinto al lenguaje en que se especificó el sistema. En el caso de TLA$^+$, especificaciones y propiedades se escriben en el mismo formalismo, con lo que no es necesario traducir de uno a otro al usar un demostrador de teoremas o un model-checker.

El sistema de pruebas de TLA$^+$ TLA$^+$ fue extendido hacia un lenguaje de pruebas formal, declarativo y jerárquico llamado TLA^{+2}. El Sistema de Pruebas TLAPS (TLA Proof System) es un framework basado en la aplicación Proof Manager (PM) que toma una especificación junto con pruebas de sus propiedades escritas en TLA^{+2} e intenta certificarlas, es decir, generar el texto de una prueba (el certificado).

Las pruebas se estructuran en forma de árbol donde por cada hoja del árbol, correspondiente al nivel más bajo de la prueba, PM genera una obligación de prueba que es enviada a uno o más demostradores secundarios para que sea verificada. Estos son los que producen por cada prueba verificada una traza o script de la prueba, que luego debe certificarse en el entorno lógico como el asistente de pruebas genérico Isabelle. De esta forma, el núcleo del entorno lógico es el único componente confiable del sistema de pruebas.

El entorno lógico debe poder interpretar fórmulas con la sintaxis y semántica de TLA$^+$. Es por esto que se está llevando a cabo la formalización de TLA$^+$ en Isabelle, llamada Isabelle/TLA$^+$[1]. Entre los demostradores secundarios que forman parte de TLAPS hasta ahora, se encuentran Zenon y el mismo Isabelle/TLA$^+$.

Estado del arte Existen formalizaciones de TLA$^+$ sobre las lógicas-objeto Isabelle/ZF [Kal95] e Isabelle/HOL [Mer97], donde HOL (High Order Logic, o lógica de alto orden) es un λ-cálculo simplemente tipado. La particularidad de TLA$^+$ con respecto a otros lenguajes de especificación es que TLA$^+$ es no-tipado, lo que hace que su sintaxis sea menos restrictiva y, por lo tanto, un lenguaje mucho más expresivo. Esto, en cambio, dificulta la verificación automática y su formalización sobre otras teorías existentes y más desarrolladas como las dos mencionadas. Isabelle/ZF parece ser una opción más razonable sobre la que basar TLA$^+$ ya que ambas están basadas a su vez sobre lógica de predicados y teoría de conjuntos. Su sistema de tipos es el tradicional de la lógica de primer orden: usa los tipos *i* (el tipo de los individuos) y *o* (el tipo de las proposiciones). Por lo tanto ZF, al igual que HOL, utilizan un sistema de tipos diferente al de TLA$^+$.

Es por estas razones que se decidió formalizar TLA$^+$ como la nueva lógica-objeto Isabelle/TLA$^+$. Actualmente, se ha codificado parte del núcleo del lenguaje, incluyendo lógica proposicional y de primer orden, teoría de conjuntos básica, funciones, puntos fijos, y la construcción de números naturales. También han sido instanciados para TLA$^+$ los principales métodos de prueba semi-automáticos de Isabelle ya existentes.

[1] El último release público de TLAPS, que incluye Isabelle/TLA$^+$ y los ejemplos, puede encontrarse en [tlaa].

Objetivos El objetivo principal de este trabajo es extender Isabelle/TLA$^+$ para dar soporte a la aritmética de TLA$^+$ definida en los módulos estándar sobre números, limitándonos a los números naturales y enteros. Esto implica definir los operadores aritméticos y probar propiedades sobre ellos para aumentar el poder de razonamiento de los métodos de prueba automáticos, lo que permitirá al TLAPS certificar pruebas de especificaciones que utilicen aritmética. Luego, se analizarán los resultados mediante la verificación en Isabelle/TLA$^+$ de las invariantes de un algoritmo como caso de ejemplo, y mediante la comparación de la certificación en TLAPS del mismo algoritmo, antes y después de realizar la extensión propuesta.

Estructura del documento El presente trabajo está organizado de la siguiente manera. En el capítulo siguiente se verá el lenguaje de especificación y de pruebas TLA^{+2}, junto con su lógica subyacente y el funcionamiento del entorno de pruebas TLAPS. En el capítulo 3 se darán los detalles necesarios sobre el asistente de pruebas Isabelle, su sintaxis y sus métodos de prueba (semi)automáticos. En el capítulo 4 se verán las teorías ya formalizadas de Isabelle/TLA$^+$, sobre las que luego en los capítulos 5 y 6 se detallará la construcción de la aritmética estándar de los números naturales y enteros, respectivamente. En este último también se verá la definición de la división entera. El análisis de un caso de ejemplo se mostrará en el capítulo 7. Finalmente en el capítulo 8 se expondrán las conclusiones y posibles trabajos futuros.

Capítulo 2

El lenguaje TLA$^+$ y su entorno de pruebas

TLA$^+$ es un lenguaje de especificación basado en la Lógica Temporal de Acciones y (una variante de) la teoría de conjuntos de Zermelo-Fränkel (ZF). La Lógica Temporal de Acciones (TLA, por Temporal Logic of Actions en inglés) es usada para describir las estructuras de datos de los sistemas y algoritmos a verificar. La teoría de conjuntos ZF es el lenguaje aceptado por la mayoría de la comunidad de matemáticos como la base estándar para formalizar la matemática, y es usada en TLA$^+$ para expresar las estructuras de datos subyacentes. TLA$^+$ fue diseñado para especificar el comportamiento de alto nivel de sistemas distribuidos y concurrentes, pero también puede usarse para especificar sistemas discretos o algoritmos [Lam03].

Debido a que el razonamiento temporal es una muy pequeña parte de las pruebas de TLA, las herramientas de verificación con las que trabajamos en este proyecto no incorporan todavía la parte temporal de la lógica, por ahora sólo maneja fórmulas de acción. Sin embargo, esto no impide describir la mayor parte de los sistemas con los que se trabaja ni razonar sobre sus propiedades.

TLA$^+$ está extensamente documentado [Lam03, Mer08]. Ya que sólo estamos interesados en razonar sobre su lógica subyacente, que es relativamente conocida y simple, no se darán otros detalles de TLA$^+$ ni de sus operadores temporales más que los que se muestran a continuación. Toda la notación que no es estándar será explicada cuando sea necesario.

A continuación se describen el lenguaje TLA^{+2}, comenzando por su lógica subyacente, el lenguaje de especificación TLA$^+$ y luego la extensión a TLA^{+2} con el nuevo lenguaje de pruebas. Finalmente describimos TLAPS, un framework que verifica (y certifica) las pruebas escritas en el nuevo lenguaje.

2.1. La Lógica Temporal de Acciones

A diferencia de otras lógicas temporales, la Lógica Temporal de Acciones (TLA) [Lam94] fue diseñada para escribir especificaciones que consisten principalmente de matemática común (no temporal) con sólo un poco de lógica temporal. Es la base de TLA$^+$, un lenguaje de especificación completo.

TLA a su vez combina dos lógicas: una lógica de acciones y una lógica temporal lineal estándar. En TLA, un *estado* es una asignación de valores a las variables, y un *comportamiento* es una secuencia de estados. Las fórmulas de TLA se distinguen, precisamente, entre *fórmulas de transición*, que describen estados (por medio de predicados y funciones de estado) y transiciones de estado (las acciones), y *fórmulas temporales*, que definen comportamientos.

Básicamente, las fórmulas de transición son fórmulas comunes de la lógica de primer orden no-tipada. Un *predicado* es una expresión Booleana que contiene constantes y variables (por ejemplo, $x^2 + y - 3$) ; una *función de estado* es una expresión no-Booleana que contiene constantes y variables ($x^2 = y - 3$ y $x \in Nat$) ; y una *acción* es una expresión Booleana que contiene constantes, variables y

variables primadas ($x' + 1 = y$ y $x - 1 \notin z'$) que depende de dos estados, donde las variables no-primadas corresponden al primer estado y las primadas al estado siguiente. La cuantificación en TLA está permitida sólo sobre las constantes (también llamadas *variables rígidas* porque no cambian a través de los estados) y no sobre las variables (llamadas *variables flexibles*). La cuantificación tradicional se introduce con TLA$^+$.

Una fórmula construida con operadores constantes, constantes, variables y variables primadas es válida si y sólo si es una fórmula válida en la lógica subyacente cuando constantes, variables y variables primadas son tratadas como variables distintas de la lógica. Es decir, v y v' son consideradas como variables distintas en la lógica subyacente para cualquier variable v de TLA. Ya que cualquier fórmula de acción se puede reducir a este tipo de fórmulas, el razonamiento de acciones se puede reducir inmediatamente al razonamiento en la lógica subyacente. Por lo tanto a las variables y las variables primadas se las considerarán como constantes en las fórmulas.

La *lógica temporal* de TLA es una variante de la lógica temporal lineal de A. Pnueli y puede considerarse como un subconjunto de la lógica estándar LTL (Lógica Temporal Lineal) [MP92]. Las lógicas de tiempo lineal son apropiadas para formular propiedades que deben cumplirse en todas las ejecuciones posibles de un sistema, como la propiedad de *correctitud*, que afirma que el sistema es correcto con respecto a su especificación. En TLA las *fórmulas temporales* se construyen a partir de fórmulas elementales con operadores booleanos y el operador unario $\Box$ (se lee *siempre*). La lógica temporal permite expresar condiciones de *fairness*, que expresan qué acciones deben (eventualmente) ocurrir.

TLA es práctico para describir sistemas porque todas las complejidades de una especificación están en las fórmulas de acción, que expresan las *propiedades de seguridad*. Los operadores temporales son usados esencialmente sólo para expresar *propiedades de vivacidad*, incluyendo las condiciones de *fairness*. La mayor parte del trabajo en una prueba de TLA está en probar fórmulas de acción; el razonamiento temporal sólo se da en las pruebas de vivacidad y está limitado a la lógica temporal proposicional y a la aplicación de un conjunto de reglas cuyas premisas principales son fórmulas de acción.

Todas las fórmulas de TLA pueden expresarse en términos de los operadores matemáticos usuales (como $\wedge$) más tres operadores nuevos: $'$ (variables primadas), $\Box$ y $\exists$ (cuantificador existencial temporal). Sintácticamente, una *fórmula* en TLA tiene una de las siguientes formas básicas:

$$\begin{array}{cccccc} P & \Box[\mathcal{A}]_f & \Box F & \exists x : F & & \\ \neg F & F \wedge G & F \vee G & F \Rightarrow G & F \equiv G & \end{array}$$

donde P es un predicado, f una función de estado, $\mathcal{A}$ es una acción, x es una variable, y F y G son fórmulas de TLA. Existen otros símbolos que pueden expresarse en función de los ya mencionados (y, por supuesto, todos los operadores Booleanos pueden expresarse en función de $\neg$ y $\wedge$). Los operadores Booleanos tienen su significado usual. Existen otros símbolos con una semántica más compleja en la sintaxis de TLA (como *WF* y *SF*, sobre fairness), que se pueden expresar en función de los símbolos mencionados. (Para los operadores temporales y las definiciones precisas ver [Lam94].)

2.2. El lenguaje de especificación TLA$^+$

La lógica TLA permite describir el comportamiento dinámico de un sistema y el lenguaje de especificación TLA$^+$ le provee el azúcar sintáctico necesario para escribir especificaciones más complejas. TLA$^+$ agrega básicamente tres cosas a TLA: una lógica subyacente más poderosa, un mecanismo para definir operadores, y módulos.

Lógica subyacente La lógica subyacente de TLA$^+$, sin tener en cuenta la lógica temporal, es básicamente la teoría de conjuntos ZF más lógica clásica de primer orden no-tipada con el operador de Hilbert

ε. La mayor diferencia entre esta lógica subyacente y la tradicional ZF es que las funciones son definidas a partir de axiomas en lugar de ser representadas como conjuntos de pares ordenados.

La teoría de conjuntos de Zermelo-Fränkel es un sistema axiomático para expresar la teoría de conjuntos sin sus paradojas usuales (como la paradoja de Russell). Está basada en una signatura que consiste sólo de un predicado binario ($\in$) sin símbolos de funciones. TLA+ depende mucho del operador de elección de Hilbert[1]. De hecho, toda la lógica subyacente sin la parte temporal puede ser definida en función de los cuatro operadores primitivos $\wedge$, $\neg$, $\in$ y ε.

El operador de Hilbert ε [Lei69] es equivalente en TLA+ al operador CHOOSE. La expresión CHOOSE $x : P$ retorna algún valor x que satisfaga P, e incluso retorna un valor cuando ningún x satisface P. CHOOSE es una función, por lo tanto retorna el mismo valor cuando se aplica a iguales argumentos. Si hay más de un posible valor, la expresión está subespecificada, y el resultado puede ser cualquiera de los valores. Sin embargo, para cada interpretación, la elección del resultado es siempre determinística. De este operador es a partir del cual se definen muchos de los otros operadores de TLA+, incluidos los cuantificadores tradicionales $\exists$ y $\forall$.

Definición de operadores Se agrega en TLA+ un mecanismo para definir nuevos operadores, donde un operador definido por el usuario es esencialmente una macro que es expandida sintácticamente. (TLA+ permite definir funciones recursivamente, pero son traducidas a definiciones usando el operador CHOOSE.) Por ejemplo:

- $F(x_1, \ldots, x_n) \triangleq exp$ define el operador F tal que $F(x_1, \ldots, x_n)$ es igual a exp con cada identificador x_k reemplazado por e_k, y
- $f[x \in S] \triangleq exp$ define la función f con dominio en S tal que $f[x] = exp$ para todo x en S.

Módulos Un *módulo* permite importar definiciones y teoremas de otros módulos. Un módulo tiene como parámetros las variables y constantes declaradas, y puede ser instanciado en otro módulo al sustituir sus parámetros por expresiones.

2.2.1. Estructura general de una especificación

Las especificaciones de TLA+ están organizadas en módulos que contienen declaraciones (de parámetros), definiciones (de operadores), y afirmaciones (de hipótesis y teoremas). Cada especificación de un sistema tiene un módulo raíz. Al estar basado en la teoría de conjuntos ZF, todos los valores son conjuntos, y por lo tanto no hay declaraciones de tipo.

La forma usual de escribir una especificación de un sistema, a la que llamamos *Spec*, es:

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge Fairness_{vars}$$

donde *Next* es la disyunción de las acciones permitidas del sistema, *vars* son todas las variables del sistema, y *Fairness* es una conjunción de fórmulas de *weak fairness* (WF) y/o de *strong fairness* (SF). Informalmente, una condición de *fairness* afirma que una acción debe ocurrir eventualmente si está habilitada "con cierta frecuencia". Dependiendo de cómo se interprete qué es frecuente se expresa en las fórmulas de WF o de SF. El primer estado de cada ejecución debe satisfacer el predicado *Init* y cualquier par de estados de una ejecución deben estar relacionados por una de las acciones definidas en *Next*. Estas dos fórmulas, complementadas por la condición *Fairness*, especifican el comportamiento del sistema.

[1] Adoptar el operador de Hilbert ε implica agregar el llamado *axioma de elección* a ZF; por lo tanto, en realidad estamos usando la teoría de conjuntos llamada ZFC (la C es por *axiom of Choice*).

2.2.2. Teoremas e invariantes

El lenguaje TLA$^+$ permite expresar premisas del sistema con la palabra clave ASSUME y teoremas sobre propiedades del sistema que deban cumplirse con la palabra clave THEOREM. Estas expresiones también son usadas eventualmente por las herramientas de verificación automática, como el model-checker de TLA$^+$ [Lam03, cap. 14] o el entorno de pruebas TLAPS que veremos a continuación.

Un sistema descripto por la fórmula *Spec* tiene la propiedad *Prop* si y sólo si cada comportamiento que satisface *Spec* también satisface *Prop*, es decir, si y sólo si $Spec \Rightarrow Prop$ es válida. Normalmente se quieren probar tres tipos de propiedades: las invariantes de tipo (expresan que las variables pertenecen al conjunto de "tipos" correcto), las propiedades de seguridad (las invariantes del sistema) y las propiedades de vivacidad o *fairness*. Hacia el final del módulo se declaran los teoremas que opcionalmente deberían ser probados.

Las *invariantes* caracterizan el conjunto de estados que se pueden alcanzar en toda la ejecución de un sistema. Son la forma básica de las propiedades de seguridad y el punto de partida para cualquier forma de verificación de un sistema. En TLA, una invariante se expresa con una fórmula de la forma $\Box I$, para una fórmula de estado I. Por ejemplo, usualmente se declaran los teoremas

THEOREM $Spec \Rightarrow \Box TypeInvariant$
THEOREM $Spec \Rightarrow \Box Safety$
THEOREM $Spec \Rightarrow Liveness$

donde la fórmula *TypeInvariant* es una invariante de tipos, *Safety* es la propiedad de seguridad y *Liveness* la propiedad de vivacidad. La declaración de una invariante de tipos no restringe a las variables que afecta, sólo declara una fórmula. Y el teorema afirma que la especificación respeta esta propiedad.

La expresión ASSUME P afirma que P es verdadera en el módulo, donde las términos de P son sólo constantes. Por ejemplo, ASSUME $(N \in Nat) \land (N > 0)$. Estas afirmaciones tampoco tienen efecto sobre las definiciones de la especificación; sin embargo, forman parte de las hipótesis al momento de probar los teoremas del módulo.

2.2.3. Los módulos estándar sobre números

TLA$^+$ cuenta con una librería de módulos estándar [Lam03, cap. 18]. Estos módulos facilitan la lectura y escritura de las especificaciones que los usan, y son usados por las herramientas de verificación. La librería cuenta con los módulos *Sequences*, *FiniteSets*, *Bags*, *RealTime* y los módulos sobre números, que son los que interesan en este trabajo: *Peano*, *Naturals*, *Integers*, *Reals* y *ProtoReals*.

En TLA$^+$ los números están predefinidos, incluso en un módulo que no extiende o instancia uno de los módulos de números estándar. Sin embargo, no lo están los conjuntos de números como *Nat* y operadores aritméticos como +. Con respecto a la semántica de una representación de un número natural se define de la manera usual:

$$0 \triangleq \text{ZERO} \qquad 1 \triangleq \text{SUCC}[\text{ZERO}] \qquad 2 \triangleq \text{SUCC}[\text{SUCC}[\text{ZERO}]] \qquad \ldots$$

Debido a que se puede dar que un módulo extienda indirectamente los módulos *Naturals* y *Reals* al mismo tiempo, la definición de operadores como + debe ser la misma para los naturales y para los reales. El módulo *Naturals* define los siguientes operadores y conjuntos:

$+$	$*$	$<$	$\leq$	Nat	$\div$ [división entera]
$-$ [menos binario]	$\hat{}$ [exponenciación]	$>$	$\geq$	..	% [módulo]

La división entera ($\div$) y el módulo (%) se definen de manera que las dos condiciones siguientes se cumplan, para cualquier entero a y entero positivo b:

$$a \,\%\, b \in 0 \,..\, (b-1) \qquad a = b * (a \div b) + (a \,\%\, b)$$

El módulo *Integers* extiende el módulo *Naturals* y además define el conjunto *Int* de enteros y el menos unario (–.). El módulo *Reals* extiende a *Integers* e introduce el conjunto *Real* de números reales y la división común (/). En matemática, a diferencia de los lenguajes de programación, los enteros son números reales. Por lo tanto, *Nat* es un subconjunto de *Int*, que es un subconjunto de *Real*.

El conjunto *Nat* de los números naturales, el elemento ZERO y la función sucesor SUCC están definidos en el módulo *Peano*, que simplemente establece que estos elementos satisfacen los axiomas de Peano.

El módulo *ProtoReals* extiende al módulo *Peano* y define los números reales usando el conocido resultado matemático que establece que los reales son definidos como el único *campo ordenado completo*[2]. Lo hace de la siguiente manera: se define un modelo en el que dados un conjunto *R*, dos operaciones *Plus* y *Times* y una relación *Leq* satisfacen las propiedades elementales de un campo ordenado completo. Luego define el conjunto *Reals*, las operaciones + y $*$, y la relación $\leq$ como una instancia (usando el operador CHOOSE) de *R*, *Plus*, *Times* y *Leq* respectivamente. Aquí también se define el conjunto de los enteros como $Int \triangleq Nat \cup \{Zero - n : n \in Nat\}$.

En los módulos *Naturals* e *Integers*, los operadores ya mencionados y los conjuntos *Nat* e *Int* son una instancia de los operadores definidos en *ProtoReals*. En Isabelle/TLA+, no los definiremos axiomáticamente de esta manera, sino que los definiremos recursivamente y luego probaremos las propiedades que los definen.

2.3. El lenguaje de pruebas de TLA+2

El lenguaje TLA+ ha sido extendido a un nuevo lenguaje llamado TLA+2. La característica sobresaliente de TLA+2 es su lenguaje de pruebas jerárquico que permite la verificación de especificaciones de sistemas (en conjunto con herramientas como TLAPS que se describe en la sección siguiente). Siempre fue posible expresar propiedades sobre la correctitud de sistemas en TLA+ (con THEOREM), pero no escribir sus pruebas. Los constructores de pruebas de TLA+2 fueron creados basados en el estilo jerárquico para escribir pruebas informales dado en [Lam93].

El objetivo del lenguaje es escribir pruebas fáciles de leer y escribir por alguien que no tenga conocimientos de cómo se chequean las pruebas. Esto ha llevado a desarrollar un lenguaje en su mayor parte declarativo, construido en torno al uso y prueba de afirmaciones en lugar de la aplicación de tácticas para la búsqueda de la prueba. Es por lo tanto más cercano a Isabelle/Isar [Nip] que a lenguajes interactivos y operacionales como Vernacular de Coq [Typ08].

En cualquier punto de una prueba de TLA+ hay una *obligación de prueba* que debe ser probada. La obligación tiene un *contexto* de hipótesis conocidas, definiciones, declaraciones, y un *objetivo* que se quiere probar.

Estructura de una prueba en TLA+2

Un teorema de TLA+ es seguido opcionalmente de una prueba. Una prueba comienza con la palabra clave PROOF (que es opcional) y tiene una de dos formas: puede ser una prueba terminal, o una secuencia

[2] Un *campo* es una estructura algebraica que cuenta con dos operaciones (suma y multiplicación) que cumplen con las propiedades asociativa, conmutativa y distributiva, además de la existencia de los elementos identidad e inverso de cada operación. Es ordenado si cuenta con una relación de orden total. Un campo ordenado es completo si cada subconjunto acotado por arriba tiene una mínima cota superior.

de pasos intermedios terminada por la palabra clave QED. Si los pasos intermedios son afirmaciones, requieren de una nueva prueba (una *subprueba*) en el siguiente nivel de la jerarquía. En otros pasos se hacen declaraciones o definiciones (con DEFINE o SUFFICES, por ejemplo) o se cambia el objetivo y no se requiere una prueba.

Una *prueba terminal* consiste básicamente de la palabra clave OBVIOUS o BY, y es el nivel más bajo de la prueba. En este punto, la prueba OBVIOUS afirma que el objetivo actual se deduce fácilmente del contexto actual. La prueba BY $e_1, \ldots, e_m$ DEFS $o_1, \ldots, o_n$ afirma que el objetivo actual se deduce fácilmente del contexto actual más las hipótesis e_i, que a su vez deben deducirse fácilmente del contexto, y de las definiciones conocidas o_j. Si ninguna de estas pruebas da resultado, suponiendo que el contexto es el adecuado, entonces se debe agregar un nivel a la prueba y probar más pasos intermedios necesarios para agregarlos al contexto. Que un objetivo se deduzca fácilmente de hipótesis y definiciones depende de qué herramientas estén intentando probarlo, como se verá en la siguiente sección.

```
THEOREM T ≜ ⟨ fórmula ⟩
PROOF
⟨1⟩1.  ...
  PROOF
    ⟨2⟩1. DEFINE T ≜ ...
    ⟨2⟩2.  ...
      ⟨9⟩1.  ...
        OBVIOUS
      ⟨9⟩2.  ...
        PROOF OMITTED
    ⟨2⟩14.  QED
            BY ⟨2⟩1, ⟨2⟩2
⟨1⟩2.  ...
  BY  ...
⟨1⟩3. QED
```

Figura 2.1: Estructura de una prueba en TLA^{+2}.

La figura 2.3 muestra como ejemplo el esquema de la prueba de un teorema identificado como *T*, donde se omiten las fórmulas. Aquí, por ejemplo, el nivel 1 de esta prueba tiene tres pasos llamados ⟨1⟩1, ⟨1⟩2 y ⟨1⟩3. El paso ⟨1⟩1 tiene a su vez una prueba de tres pasos de nivel 2 llamados ⟨2⟩1, ⟨2⟩2 y ⟨2⟩14. El paso ⟨9⟩2 tiene una prueba terminal que indica que en este caso la prueba se omite.

2.4. El Sistema de Pruebas TLAPS

El Sistema de Pruebas TLAPS (TLA Proof System) es una plataforma para la verificación mecánica de pruebas de TLA^{+2} [CDLM10]. Los componentes principales son el Administrador de Pruebas TLAPM (TLA Proof Manager) y los *demostradores secundarios*[3] que este invoca. La arquitectura general del Sistema de Pruebas TLAPS puede verse en la figura 2.4. (Se describe en detalle en [CDLM08].) A continuación veremos paso por paso el proceso que lleva a cabo.

En primer lugar, el usuario ingresa al TLAPM un archivo con la especificación de un sistema en TLA^{+2}, incluyendo pruebas de sus teoremas. TLAPM toma el módulo raíz, expande recursivamente los módulos que este importa y se deshace de toda la estructura de módulos, ya que los constructores para importar módulos no son soportados por los demostradores secundarios. También se expanden las

[3] El término en inglés es *back-end provers*, cuya traducción literal es *demostradores de fondo*.

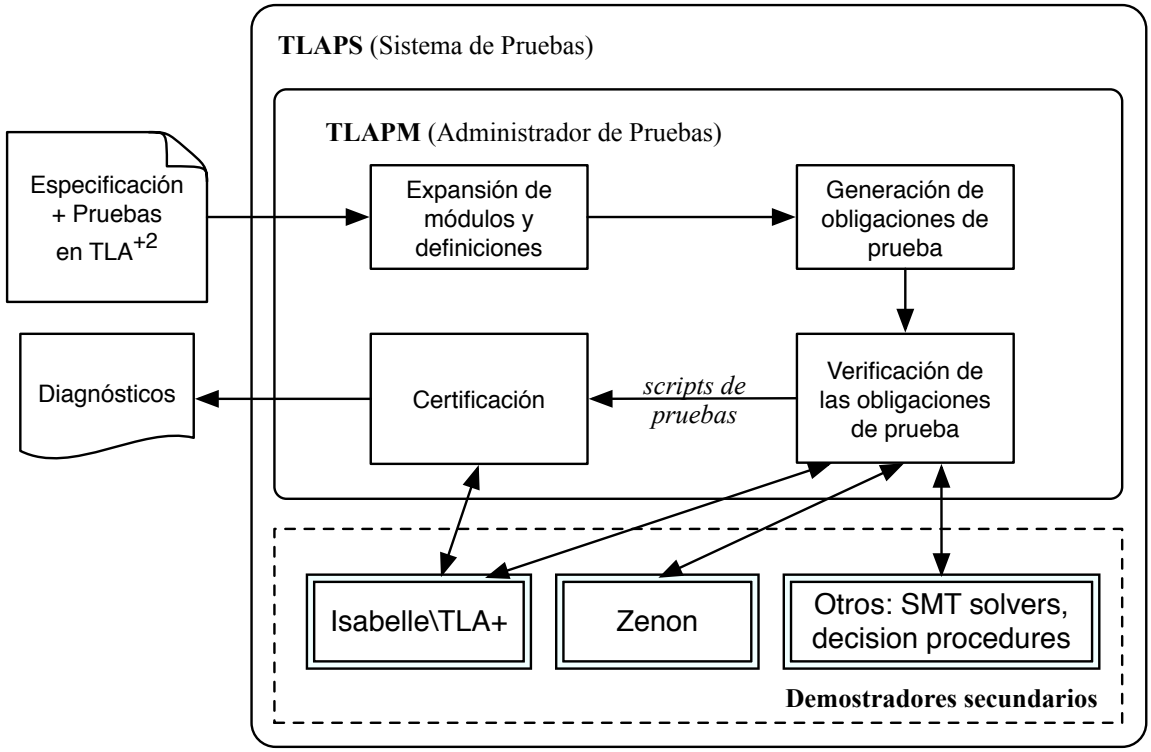

Figura 2.2: Arquitectura general del entorno de pruebas TLAPS.

definiciones y el operador ’ (que prima variables) sobre las expresiones, y se llevan todas las variables, inclusive las primadas, al nivel de las constantes que, como se mencionó anteriormente en 2.1, no modifica la validez de la fórmula.

En el siguiente paso, el TLAPM genera una *obligación de prueba* por cada prueba terminal (es decir, las hojas del árbol de pruebas) y organiza la ejecución de los demostradores secundarios para que las verifiquen. Cada obligación de prueba es independiente y puede ser probada por separado. Cuando los demostradores secundarios no pueden encontrar una prueba en un límite razonable de tiempo, el sistema informará cuál obligación es la que falló, junto con su contexto y objetivo de la prueba. El usuario debe entonces determinar si falló debido a que depende de hechos o definiciones ocultas, o si el objetivo es demasiado complejo y necesita ser refinado con otro nivel de prueba.

Si un demostrador secundario encuentra una prueba de una obligación, es decir si la puede verificar, genera una traza de la prueba realizada, llamada *script de prueba*. El Administrador de Pruebas PM mediará la *certificación* de los scripts de prueba en un entorno lógico confiable, que en el diseño actual es Isabelle/TLA+[4]. (La lógica-objeto Isabelle/TLA+ se verá en detalle en el capítulo 4.) Por lo tanto es preferible que los demostradores secundarios generen scripts de prueba en un lenguaje de pruebas que Isabelle acepte, es decir, en el lenguaje Isar (ver sección 3.3.2). En el momento en que un script es generado, se lo pasa a Isabelle/TLA+ para su certificación. Y al mismo tiempo, en paralelo, las restantes obligaciones de prueba comenzarán a ser verificadas por los demostradores secundarios.

Hasta el momento, los demostradores secundarios soportados por TLAPS son Zenon y el mismo Isabelle/TLA+ (que no sólo certifica los scripts de prueba). El Administrador de Pruebas envía una obligación de prueba primero a Zenon. Si Zenon puede probarla, produce un script en lenguaje Isar, que luego es enviado por el Administrador a Isabelle para su certificación. Si Zenon falla, el Administrador

[4]TLAPM está diseñado genéricamente y puede soportar otros entornos similares a Isabelle.

envía la obligación de prueba a Isabelle para que intente probarla con sus tácticas automáticas.

La arquitectura del Sistema de Pruebas se adapta fácilmente a otros demostradores secundarios. Aunque siempre es preferible que las pruebas de los demostradores secundarios puedan ser certificadas, se permite el uso también de demostradores que producen pruebas no-certificables en teorías importantes como la aritmética. Si uno de estos demostradores es usado, TLAPS no provee la misma certeza matemática que con una prueba formal completa. De hecho, TLAPS no requiere siquiera que las pruebas estén terminadas antes de que pueda comenzar a verificarlas. Esto no sólo es esencial al escribir pruebas, sino también necesario porque el lenguaje TLA$^+$ todavía no está soportado completamente (en especial lo relacionado con el razonamiento temporal)

Finalmente, luego de certificar todas las obligaciones de prueba posibles generadas por las pruebas terminales, se procede a certificar el teorema en sí, en un proceso de dos pasos. Primero, el PM genera la estructura de un lema (y sus pruebas en Isabelle/TLA$^+$) que establece simplemente que la colección de obligaciones de pruebas terminales implican el teorema. Luego, el PM genera una prueba del teorema usando la estructura del lema y las obligaciones ya certificadas. Si Isabelle acepta la prueba, estamos seguros de que la versión traducida del teorema es verdadera en Isabelle/TLA$^+$, sin tener en cuenta cualquier error que pudo haber cometido el PM.

Suponiendo que Isabelle/TLA$^+$ es correcto, una vez que haya certificado un teorema sabemos que un error es posible sólo si el PM tradujo incorrectamente a Isabelle/TLA$^+$ la declaración del teorema. Sin embargo, el Sistema de Pruebas TLAPS, incluyendo Isabelle/TLA$^+$, está enfocado en evitar errores de alto nivel en sistemas, no en dar los fundamentos formales de la matemática. Es mucho más probable que un error en el sistema sea debido a una especificación incorrecta o incompleta que a una prueba incorrecta que haya sido aceptada inadvertidamente por errores en TLAPS.

Zenon y otros demostradores secundarios

Actualmente, el único demostrador cuyas pruebas pueden ser certificadas es Zenon. TLAPS puede llegar a usar en el futura otros demostradores secundarios cuyas pruebas puedan ser certificadas por Isabelle/TLA$^+$, incluyendo a SMT solvers como veriT [BCBdODF09] y ciertas implementaciones de decision procedures para aritmética de Presburger [CN08].

Zenon [BDD07] es un demostrador de teoremas automático para lógica clásica de primer orden (con igualdad), basado en el método tableau. Fue inicialmente diseñado para generar pruebas chequeables en el demostrador de teoremas Coq. Luego fue extendido para generar scripts de pruebas chequeables en Isabelle. Uno de sus objetivos de diseño es resolver problemas simples de manera rápida.

También se ha extendido Zenon para interpretar la lógica de TLA^{+2}, incluyendo las definiciones y reglas sobre conjuntos y funciones. El soporte para nuevas teorías (como strings, tuplas, records y lógica temporal) se van agregando una vez que la correspondiente teoría haya sido desarrollada en Isabelle/TLA$^+$, para asegurar que Zenon produzca scripts de pruebas que Isabelle/TLA$^+$ pueda chequear.

Capítulo 3

El asistente de pruebas Isabelle

Isabelle [isa] es un demostrador de teoremas genérico, diseñado para hacer razonamiento interactivo en una variedad de teorías formales. Provee un lenguaje para describir las lógicas-objeto y para probar sus teoremas, y procedimientos de pruebas útiles para varias lógicas.

En los demostradores de teoremas genéricos, se llama *lógica-objeto* a la codificación o axiomatización de una lógica formal propiamente dicha, es decir, un sistema formal o cálculo lógico que consiste de un lenguaje formal y un conjunto de reglas de inferencia. Isabelle incluye una amplia familia de lógicas-objeto. Algunas son constructivas y otras clásicas. Otras están basadas en conjuntos, otras en tipos y funciones, y otras en dominios. En Isabelle se han formalizado, por ejemplo, lógicas de primer orden (FOL), teoría de conjuntos de Zermelo-Fräenkel (ZF), lógica de alto orden (HOL) y teoría de tipos constructiva (CTT de Per Martin-Löf).

Isabelle es un descendiente del demostrador de teoremas LCF (Logic for Computable Functions) y fue escrito originalmente en el lenguaje ML [1]. Las reglas de inferencia y tácticas de pruebas a bajo nivel de Isabelle se escriben en este lenguaje. Isabelle representa las reglas como proposiciones (no como funciones) y construye las pruebas combinando reglas. Estas operaciones constituyen la meta-lógica (o entorno lógico) en la cual se formalizan las lógicas-objeto. Una buena introducción a Isabelle son [NPW02] y [Pau04].

3.1. Isabelle/Pure: la meta-lógica

Las lógicas-objeto se construyen sobre la lógica llamada Isabelle/Pure, que implementa la meta-lógica y provee las estructuras de datos fundamentales: tipos, términos, signaturas, teoremas y teorías, tácticas y *tacticals* (combinadores de tácticas) [Pau89].

Las *reglas de inferencia* son teoremas de la *meta-lógica* de Isabelle, que es un fragmento de una lógica de alto orden *intuicionista*. Las reglas de inferencia tienen la forma:

$$\frac{P_1 \qquad \ldots \qquad P_n}{Q}\ (R) \tag{3.1}$$

donde las fórmulas P_i son las premisas, la fórmula Q es la conclusión y R es el nombre de la regla. Su notación en Isabelle es:

$$[\![P_1;\ldots;P_n]\!] \Longrightarrow Q.$$

Las fórmulas P_i y Q son proposiciones de la meta-lógica. Las proposiciones atómicas afirman una declaración sobre una lógica-objeto, como 'P es verdadera', 'A es un tipo bien formado', o 'a tiene tipo

[1] De hecho, el lenguaje de programación ML (Meta Language) fue creado para escribir las tácticas de prueba de LCF.

A'. Estas generalmente incluyen predicados declarados por el usuario. Las proposiciones se combinan por medio de los *meta-conectores*, o conectores del nivel de la meta-lógica: $\Longrightarrow$, $\bigwedge$ y $\equiv$. Estos símbolos fueron elegidos para diferenciarlos de los símbolos de las lógicas-objeto.

El conector $\Longrightarrow$ es la meta-implicación. Expresa una consecuencia lógica en las reglas, $P \Longrightarrow Q$ significa 'P implica Q'. La notación $[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$ abrevia a

$$P_1 \Longrightarrow (\cdots \Longrightarrow (P_n \Longrightarrow Q) \cdots)$$

El conector $\bigwedge$ es el meta-cuantificador universal. Expresa generalidad en las reglas, $\bigwedge x.\ P$ significa 'ϕ es verdadera para todo x'. Las variables meta-ligadas localmente formalizan la condición 'x no es libre en ...' típicas de las reglas de los cuantificadores.

El conector $\equiv$ es la meta-igualdad. Expresa definiciones; $a \equiv b$ significa 'a es igual a b'.

La meta-lógica incluye una regla para $\equiv$, que expande las definiciones explícitamente. Aunque hay otras reglas para $\Longrightarrow$ y $\bigwedge$, estos conectores están codificados implícitamente en las reglas de resolución de Isabelle.

3.2. Sintaxis básica

Trabajar con Isabelle implica crear teorías. En términos generales, una teoría es una colección de tipos, funciones y teoremas, análogo a un módulo de un lenguaje de programación o una especificación en un lenguaje de especificación. Un conjunto de teorías forma una lógica-objeto. La estructura general de una teoría se puede ver en la figura 3.1. El comando **theory** declara la teoría de nombre T y el comando **imports** importa recursivamente las definiciones y teoremas de las teorías ya existentes $B_1, \ldots, B_n$ de las que depende. En la teoría T se declaran nuevos tipos, se definen funciones, operadores, axiomas, etc. y se prueban teoremas sobre ellos. Todo lo definido en las teorías importadas es automáticamente visible en la teoría actual.

theory T
imports B_1 ... B_n
begin
...
declaraciones, definiciones y pruebas
...
end

Figura 3.1: Estructura general de una teoría en Isabelle

Declaración de tipos y constantes

En la lógica-objeto de TLA$^+$, al ser un lenguaje no-tipado, hay un solo tipo declarado, el tipo c. Los objetos de aridad 0 tienen tipo c, las fórmulas de aridad 1 tienen tipo $c \Rightarrow c$, etc.

Las constantes de nombre $c_1, \ldots, c_n$ y tipos $\tau_1, \ldots, \tau_n$ se declaran con el comando **consts**. A las constantes y a las definiciones, opcionalmente, se les puede dar entre paréntesis una notación distinta.

consts $c_1 :: \tau_1$ (notación)
$\vdots$
$c_1 :: \tau_n$ (notación)

Definiciones

A las constantes se les puede dar una *definición* que es simplemente una abreviación, es decir, dar un nuevo nombre para una construcción ya existente. En particular, las definiciones no pueden ser recursivas. Isabelle ofrece definiciones en el nivel de tipos y términos. Aquellas en el nivel de tipo son llamadas *sinónimos de tipos* y aquellas en el nivel de términos son llamadas simplemente definiciones. Las definiciones se hacen con el comando **definition** y tienen la forma:

definition *f* (notación) **where** "$f(x_1, \ldots, x_n) \equiv t$"

donde f es el nombre de la constante y $x_1 \ldots x_n$ son los argumentos, sobre los que no está permitido hacer pattern-matching. El símbolo ≡ es una forma especial de igualdad que debe usarse en definiciones de constantes. El nombre por defecto de cada definición es *f_def*, donde f es el nombre de la constante definida. El comando **constdefs** declara el tipo de la constante y da la definición al mismo tiempo.

constdefs $f :: \tau$ (notación) "$f(x_1, \ldots, x_n) \equiv t$"

El comando **axioms** declara fórmulas como teoremas de la meta-lógica, donde $r_1, \ldots, r_n$ son los identificadores de las fórmulas $P_1, \ldots, P_n$:

axioms r_1 [*atributos*]: P_1
$\vdots$
r_1 [*atributos*]: P_n

De hecho, los axiomas son "teoremas axiomáticos" y se puede referir a ellos como cualquier otro teorema. Los *atributos* le dicen a Isabelle qué tipo de regla es y cómo pueden usarla los métodos de prueba automáticos (ver sección 3.4). Los atributos más comunes son: *simp* que la declara como regla del Simplificador, *intro* y *elim*, que la declaran como regla de introducción o eliminación. A su vez, estos atributos si son seguidos de ! pueden declararse como seguros, lo que significa que no transforman la prueba en una que no se pueda probar.

Teoremas

Los *teoremas* son reglas que, a diferencia de los axiomas, requieren una prueba. En la figura 3.2 se muestra la declaración de un teorema seguido de su prueba. El comando **theorem** (que es equivalente a **lemma**) le da al teorema el nombre T y establece que se debe probar la conclusión C dadas las hipótesis H_i. Las construcciones

assumes $h_1 : H_1$ **and** $\ldots$ **and** $h_n : H_n$ **shows** C y $[\![H_1; \ldots; H_n]\!] \Longrightarrow C$

son equivalentes, con la diferencia que en la segunda no se le puede dar los nombres h_i a las premisas. Los atributos son los mismos que para los axiomas.

A continuación se nombran algunos de los comandos de pruebas más comunes. El comando **unfolding** *f_def* reemplaza a f por su definición en el estado de la prueba. Si no quedan más subobjetivos, se termina la prueba con el comando **done**. También se puede terminar una prueba con el comando **by** (*método/s*), que es similar a hacer **apply** (*método/s*, **assumption**) y luego **done**. El comando **apply** (*método/s*) aplica métodos de razonamiento automático (ver sección 3.4), o métodos de prueba de deducción natural, que es uno de los dos estilos de prueba de Isabelle. El otro estilo es el del lenguaje de pruebas Isar (ver sección 3.3.2), que utiliza los comandos **proof** y **qed** para delimitar sus pruebas. En las siguientes dos secciones se muestran los dos estilos de prueba de Isabelle.

theorem / **lemma** *T* [*atributos*]*:*
assumes H_1 **and** ... **and** H_n **shows** *C*
⋮
unfolding *definición/es*
apply *(método/s)*
⋮
proof
prueba en lenguaje Isar
qed
done / **by** *(...)*

Figura 3.2: Estructura de la declaración y prueba de un teorema en Isabelle

3.3. Pruebas

Una regla de inferencia como 3.1 puede leerse de dos maneras. Una es de la forma: "si conozco esto y esto entonces puedo deducir esto". Por ejemplo, si tengo la prueba de A y la prueba de B entonces tengo la prueba de $A \wedge B$. A esto se lo llama razonamiento "hacia adelante" o *top-down*, desde las premisas a la conclusión. La otra manera es decir: "para probar esto, tengo que probar esto y esto". Por ejemplo, para probar $A \wedge B$, hay que probar A y hay que probar B. Esto es llamado razonamiento "hacia atrás" o bottom-up, que procede de la conclusión a las premisas. Se dice que la conclusión es el *objetivo* a probar y las premisas son los *subobjetivos*. Por eso se la llama también prueba dirigida por el objetivo.

Isabelle implementa dos estilos de prueba: uno basado en deducción natural y otro, implementado por el lenguaje Isar, que utiliza sólo el razonamiento hacia adelante y que tiene por objetivo escribir pruebas fáciles de leer para las personas, más que para las computadoras.

3.3.1. Pruebas por deducción natural

La deducción natural es un intento de formalizar la lógica de manera que refleje los patrones de razonamiento humano [Pau86]. Para cada símbolo lógico hay dos tipos de reglas: de introducción y de eliminación. La regla de introducción permite inferir un símbolo lógico, llevan la información hacia adelante por construcción. La regla de eliminación permite deducir consecuencias de un símbolo lógico, llevan la información hacia atrás por deconstrucción. Por lo tanto, en el modelo formal de *deducción natural*, el flujo de información es bidireccional. Una prueba de deducción natural no tiene una lectura puramente hacia adelante o hacia atrás, lo que hace que sean fácil de usar en una prueba interactiva, pero difícil de automatizar.

Las *tácticas* son funciones que transforman estados de prueba. En las pruebas de Isabelle se razona principalmente hacia atrás, aplicando tácticas sucesivas a los (sub)objetivos desde el estado de prueba inicial hasta llegar a un estado de prueba final. En el modelo de deducción natural, cada paso de la prueba consiste en identificar el símbolo lógico más externo de una fórmula y aplicar la correspondiente regla. El resultado es la creación de nuevos subobjetivos a partir de la unificación con la fórmula elegida.

Al expandir las definiciones de la constantes puede hacer aumentar enormemente el largo del objetivo. Derivar las reglas de deducción natural para tales constantes permite razonar en términos de sus propiedades clave, lo que de otra manera pueden llegar a ocultarse por los detalles de sus definiciones.

Reglas de introducción

Una regla de introducción permite inferir una fórmula que contiene un símbolo lógico determinado. Por ejemplo, la regla de introducción de la conjunción ($\wedge I$) dice que si tenemos P y Q como premisas entonces tenemos $P \wedge Q$. Análogamente, para la disyunción, hay dos reglas de introducción ($\vee I1$ y $\vee I2$). La notación usual es:

$$\frac{P \qquad Q}{P \wedge Q}\ (\wedge I) \qquad \frac{P}{P \vee Q}\ (\vee I1) \qquad \frac{Q}{P \vee Q}\ (\vee I2)$$

Las reglas introducen el símbolo en su conclusión. Cuando aplicamos esta regla en una prueba, el objetivo ya tiene la forma de una conjunción. Lo que hace este paso de la prueba es sacar el símbolo del objetivo. En Isabelle estas reglas se llaman *conjI*, *disjI1* y *disjI2* y su notación es:

$$[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q \quad (conjI) \qquad ?P \Longrightarrow ?P \vee ?Q \quad (disjI1) \qquad ?Q \Longrightarrow ?P \vee ?Q \quad (disjI2)$$

Si se aplica la regla hacia atrás, Isabelle trata de *unificar* el objetivo que se quiere probar con la conclusión de la regla. Si la unificación da resultado, se deben probar los nuevos subobjetivos que corresponden a las premisas (dos en el caso de la introducción). (El proceso de unificación se explica más abajo.)

Reglas de eliminación

Las *reglas de eliminación* funcionan de manera opuesta a las reglas de introducción. Para la conjunción, hay dos reglas de eliminación. De $P \wedge Q$ se deduce P y también se deduce Q:

$$\frac{P \wedge Q}{P}\ (\wedge E1) \qquad \frac{P \wedge Q}{Q}\ (\wedge E2)$$

Para el caso de la disyunción, la situación es distinta. De $P \vee Q$ no se puede concluir que P es verdadera o que Q es verdadera. No hay una regla de eliminación directa como para la conjunción; en su lugar la regla funciona de manera indirecta. Si se intenta probar otra fórmula, por ejemplo R, y se sabe que $P \vee Q$ se cumple, entonces hay que considerar dos casos. Se puede asumir que P es verdadera y probar R y después asumir que Q es verdadera y probar R una segunda vez. Hay que probar R dos veces, bajo premisas diferentes. La notación de la regla es:

$$\frac{P \vee Q \qquad \begin{array}{c}[P]\\ \vdots\\ R\end{array} \qquad \begin{array}{c}[Q]\\ \vdots\\ R\end{array}}{R}\ (\vee E)$$

Las premisas $[P]$ y $[Q]$ se escriben entre corchetes para recalcar que son locales a sus subpruebas. En Isabelle las reglas se llaman *conjE1*, *conjE2* y *disjE* y la notación es:

$$[\![?P \wedge ?Q]\!] \Longrightarrow ?P \quad (conjE1) \qquad [\![?P \wedge ?Q]\!] \Longrightarrow ?Q \quad (conjE2)$$
$$[\![?P \vee ?Q; ?P \Longrightarrow ?R; ?P \Longrightarrow ?Q]\!] \Longrightarrow ?R \quad (disjE)$$

Unificación y métodos de aplicación de reglas de inferencia

Además de distinguir, como es habitual, entre variables libres y variables ligadas, Isabelle tiene un tercer tipo de variable, llamada *variable esquemática* o *desconocida*[2]. Sus identificadores tienen un ? como primer caracter y son usadas durante el proceso de unificación. Una variable esquemática es una

[2]Este tipo de variables son llamadas *variables lógicas* en el lenguaje de programación Prolog.

variable libre, pero puede ser instanciada con otro término durante el proceso de prueba. Por ejemplo, el teorema $x = x$ en Isabelle se representa como $?x = ?x$, lo que significa que Isabelle puede instanciarlo arbitrariamente, a diferencia de las variables comunes, que permanecen fijas, aunque parezca contradictorio.

La *unificación* –concepto conocido por los usuario de Prolog– es hacer que dos términos coincidan de forma idéntica, posiblemente reemplazando sus variables esquemáticas por términos. En los casos más simple se hace pattern-matching, que reemplaza variables en sólo uno de los términos. El método **rule** usualmente hace coincidir la conclusión de la regla con el subobjetivo que se quiere probar. El método **assumption** hace coincidir el subobjetivo con alguna de las premisas. La unificación puede instanciar variables en ambos términos; el método **rule** puede hacer esto si el objetivo contiene variables esquemáticas. La variable es actualizada al mismo tiempo en todos los lugares que aparezca en la regla o en la prueba. Las variables esquemáticas en los objetivos representan términos desconocidos. Pueden ser completadas más adelante, a veces en varias etapas y a veces automáticamente.

Suponemos la regla de inferencia R de la expresión 3.1. Los métodos de aplicación de reglas más usados son los siguientes (y son en general variantes del método **rule**):

rule R unifica Q con el primer subobjetivo reemplazándolo por n nuevos subobjetivos: las instancias de $P_1, \ldots, P_n$. Aplica básicamente razonamiento hacia atrás y por lo tanto es apropiado para las reglas de introducción.

erule R unifica Q con el primer subobjetivo y al mismo tiempo unifica P_1 con alguna de las hipótesis. El subobjetivo es reemplazado por $n-1$ nuevos subobjetivos de pruebas de instancias de $P_2, \ldots, P_n$, ahora sin la hipótesis P_1. Este método es apropiado para las reglas de eliminación.

drule R unifica P_1 con alguna premisa, la cual después elimina. El subobjetivo es reemplazado por $n-1$ nuevos subobjetivos de pruebas de instancias de $P_2, \ldots, P_n$; cada nuevo subobjetivo tiene una premisa demás: una instancia de Q. Este método es apropiado para las reglas de destrucción.

Isabelle usa unificación de alto orden [Hue75], que trabaja con λ-cálculo tipado. El procedimiento de búsqueda que utiliza es, por lo tanto, potencialmente no-decidible en el caso general pero funciona bien en los casos simples que generalmente surgen en la práctica. En particular, se encarga del razonamiento de cuantificadores.

Pruebas por inducción

Aunque no es parte del modelo de deducción natural, otro método de prueba muy utilizado, especialmente al trabajar con números naturales, es la *inducción matemática*. En Isabelle, el método *induct* aplica el principio de inducción específico del tipo de la variable al que se esté aplicando.

$$\frac{P(0) \qquad \bigwedge x.P(x) \Longrightarrow P(x+1)}{P(n)} \ (nat\_induct)$$

Para los naturales el principio de inducción es *nat_induct*: para probar la propiedad P en la variable n (aquí implícitamente de tipo natural) debe probarse $P(0)$ y que si vale $P(x)$ entonces vale $P(x+1)$.

3.3.2. El lenguaje de pruebas Isar

Tradicionalmente, una prueba en Isabelle es una secuencia más o menos estructurada de comandos **apply** que manipulan el estado de la prueba. Para una persona que lee el texto de la prueba, sólo puede entenderla si ve los estados intermedios en los que se encuentra, cosa que generalmente no es posible. El lenguaje de pruebas *Isar* [Nip, Wen09] representa otro estilo de pruebas de Isabelle que soluciona estos problemas. Son estos aspectos lo que le permiten también mejorar el mantenimiento de las pruebas. Isar

Objetivo del estado de la prueba: "$\bigwedge x_1 \ldots x_m . [\![P_1; \ldots ; P_n]\!] \Longrightarrow C$*"*
proof
 fix $x_1 \ldots x_m$ — fija las variables esquemáticas
 assume *p1: "*P_1*"* **and** ... **and** *pn: "*P_n*"* — pi es la etiqueta de la premisa P_i
 ⋮
 from *facts* **have** *" ... "* — resultado intermedio usando los hechos *facts*
  Proof ⟨⟩
 ⋮
 show *"C"*
  Proof ⟨⟩
qed

Figura 3.3: Esquema simplificado de prueba en Isar

es similar al lenguaje de pruebas de TLA^{+2} (ver sección 2.3) aunque no está estructurado exactamente de manera jerárquica como este. En Isabelle es el estilo de prueba que priorizamos usar.

En la figura 3.3 se muestra el esquema simplificado de una prueba en Isar. En cualquier momento del estado de una prueba se puede entrar al entorno de pruebas de Isar con el comando **proof**, y finalizarla con **qed**. Luego de **assume** se deben listar las premisas y opcionalmente se las puede etiquetar para un posterior uso. Por ejemplo, se las puede usar, con **from**, en la prueba de un paso intermedio Q, con **have** Q. El comando **show** establece la conclusión del objetivo. Los pasos intermedios y la conclusión deben probarse, no importa en qué estilo de prueba. En caso de que haya más de un subobjetivo, se separan sus pruebas respectivas (es decir, la estructura **fix**-**assume**-**show**) con el comando **next**.

3.4. Razonamiento automático

Isabelle no encuentra pruebas automáticamente. Las pruebas requieren de usuarios entrenados, quienes son los que deben decidir qué lemas intermedios probar y qué herramientas o métodos aplicar. Entre estos métodos, Isabelle cuenta con varias tácticas de razonamiento automático.

Las tácticas mapean el estado de una prueba a una lista de posibles estados siguientes. Es posible entonces hacer backtracking y las táctivas pueden implementar estrategias de búsqueda tales como *depth-first*, *best-first* y *iterative deepening*]. Isabelle provee varias herramientas genéricas poderosas, entre las que se encuentran el Simplificador y el Razonador Clásico que son las más usadas.

3.4.1. El Simplificador

La *simplificación* es una táctica automática que aplica reglas de reescritura sobre un objetivo y luego intenta probar el objetivo reescrito usando una táctica dada por el usuario. Una regla de reescritura condicional sólo se aplica si la simplificación recursiva prueba la condición instanciada. La reescritura funciona no sólo por ecuaciones de igualdad, sino para cualquier relación reflexiva/transitiva con leyes de congruencia.

La simplificación es una de las herramientas centrales usadas por los demostradores de teoremas y otros sistemas similares. La herramienta en sí es llamada *Simplificador* (*simplifier* en inglés). Básicamente lo que hace es aplicar iterativamente *reescritura de términos* en base a ecuaciones (llamadas *reglas de reescritura*), reemplazando su lado izquierdo por el derecho. La palabra "reescritura" es más adecuada que "simplificación" ya que los términos no necesariamente se vuelven más simples durante el proceso.

En Isabelle, los axiomas o teoremas se habilitan para ser usados por el Simplificador al poner el modificador *[simp]* luego del nombre de la regla, de la misma manera que se declaran las reglas de deducción natural. Casi cualquier axioma o teorema puede declararse como regla de simplificación y el Simplificador lo transformará en una ecuación. Por ejemplo, el teorema $\neg P$ se convierte en $P = \text{FALSE}$, o $Q \in Nat$ en $Q \in Nat = \text{TRUE}$.

Pero debe tenerse cuidado al habilitar por defecto un teorema como regla de simplificación. Sólo deben declararse las ecuaciones donde su lado derecho sea más "simple" que el izquierdo. El proceso de simplificación puede no terminar. Por ejemplo, si las ecuaciones $P = Q$ y $Q = P$ pertenecen al mismo tiempo al conjunto de reglas del Simplificador. O en el caso de las reglas de la propiedad asociativa o distributiva, que sólo cambian la estructura de las fórmulas y pueden producir una explosión de términos si no son usadas correctamente. Es responsabilidad del usuario no incluir reglas que lleven a la no terminación del proceso.

En estos casos particulares, si se quiere que el Simplificador use un teorema *A*, se lo puede agregar explícitamente al conjunto de reglas de simplificación con el modificador *add*. A veces, es necesario deshabilitar, con el modificador *del*, una regla *B* que fue agredada por defecto. Por ejemplo:

apply *(simp add: A del: B)*

3.4.2. El Razonador Clásico

El Razonador Clásico de Isabelle es un método automático que Intenta resolver la mayor cantidad de subobjetivos posibles usando mayormente simplificación y un poco de deducción natural. Se invoca con el método *auto*, y utiliza un conjunto de tácticas automáticas. Las tácticas son genéricas, es decir, que no están restringidas a una lógica en particular, y pueden ser usadas independientemente. Por lo tanto, el Razonador Clásico se instancia de acuerdo a las características particulares de cada lógica-objeto.

Cada táctica automática toma una colección de reglas, clasificadas como de introducción o de eliminación, seguras o no-seguras. En general, las reglas seguras pueden intentar usarse sin pensar, mientras que las no seguras deben usarse con cuidado. Una regla segura no debe nunca reducir un objetivo que es posible probarlo a un conjunto de subobjetivos no-probables.

El Razonador Clásico aplica heurísticas naïves para probar los teoremas en el estilo del cálculo de secuentes. A pesar de su simplicidad, puede probar una gran variedad de problemas. No está restringido a la lógica de primer orden, pero aprovecha cada regla de la deducción natural.

La mayoría de las lógicas están basadas en la lógica de predicados, como TLA, o la contienen como un subconjunto, como en el caso de la lógica de alto orden. La demostración de teoremas en la lógica de predicados es no-decidible, pero hay muchas estrategias automáticas que pueden usarse en la práctica.

Otras métodos además de *auto* implementan variantes del Razonador Clásico. El método *force* es similar a *auto* pero se aplica a un solo objetivo. Mientras que *auto* trata de simplificar subobjetivos y probar los que son fáciles, *force* trata por todos los medios de probar uno solo. El método *safe* sólo aplica reglas seguras. El método *clarify* aplica las reglas obvias sin dividir el objetivo, y *clarsimp* es una combinación de *clarify* y *simp*. El método *blast* es superior a estos si se trata de razonar sobre fórmulas sin ecuaciones.

Capítulo 4

Teorías de Isabelle/TLA$^+$

El trabajo realizado en esta tesina está construido sobre las teorías ya implementadas de Isabelle/TLA$^+$, que describiremos en este capítulo. Como toda lógica-objeto de Isabelle, Isabelle/TLA$^+$ está construida sobre Isabelle/Pure (ver sección 3.1). En la figura 4.1 puede verse las dependencias entre las teorías. (Entre corchetes figura el nombre de la teoría en Isabelle.) Así vemos que la primer teoría desarrollada es la Lógica de Predicados, que incluye la lógica proposicional, a la que luego se le fue agregando la Teoría de Conjuntos, la definición de Funciones y Puntos Fijos, y la construcción de Números Naturales.

A partir de los números naturales se construyen las teorías desarrolladas en este trabajo (en la figura aparecen en negrita) y que se verán en los dos capítulos siguientes. Estas son: las Relaciones de Orden de los naturales, los operadores Aritméticos de suma, multiplicación y resta, la definición y los operadores aritméticos de los Números Enteros, y la División entera.

La teoría que incluye las Tuplas y Records fue desarrollada en paralelo a este trabajo por el proyecto. De ella dependen parcialmente la teoría de División, Strings y Expresiones CASE. Aquí daremos una visión general de cada una de las teorías mostrando las definiciones y lemas relevantes que serán usados más adelante.

4.1. Lógica clásica de primer orden

En esta teoría se axiomatiza la lógica clásica de primer orden, que es la base de la formalización de TLA$^+$. Además, se instancian y configuran para TLA$^+$ las herramientas y métodos de prueba automáticos, como *auto* (el Razonador Clásico), *simp* (el Simplificador) y blast. Los términos y fórmulas de TLA$^+$, en la meta-lógica, tienen el tipo único *c*, que representa el universo de las constantes.

Los símbolos y expresiones de la sintaxis de TLA$^+$ lógicas tienen la semántica usual. En esta teoría se definen los que se listan a continuación (por orden de definición).

Igualdad

$=$
LET $d_1 \triangleq e_1 \ldots d_n \triangleq e_n$ IN e

Los axiomas de igualdad son reflexivos y una regla afirma que términos iguales son intercambiables en el meta-nivel (esto se usa para setear el mecanismo de reescritura).

Lógica proposicional

TRUE FALSE IF p THEN e_1 ELSE e_2 [e_1 si p es verdadero, si no e_2]
$\neg \quad \wedge \quad \vee \quad \Rightarrow \quad \Leftrightarrow \quad \neq$

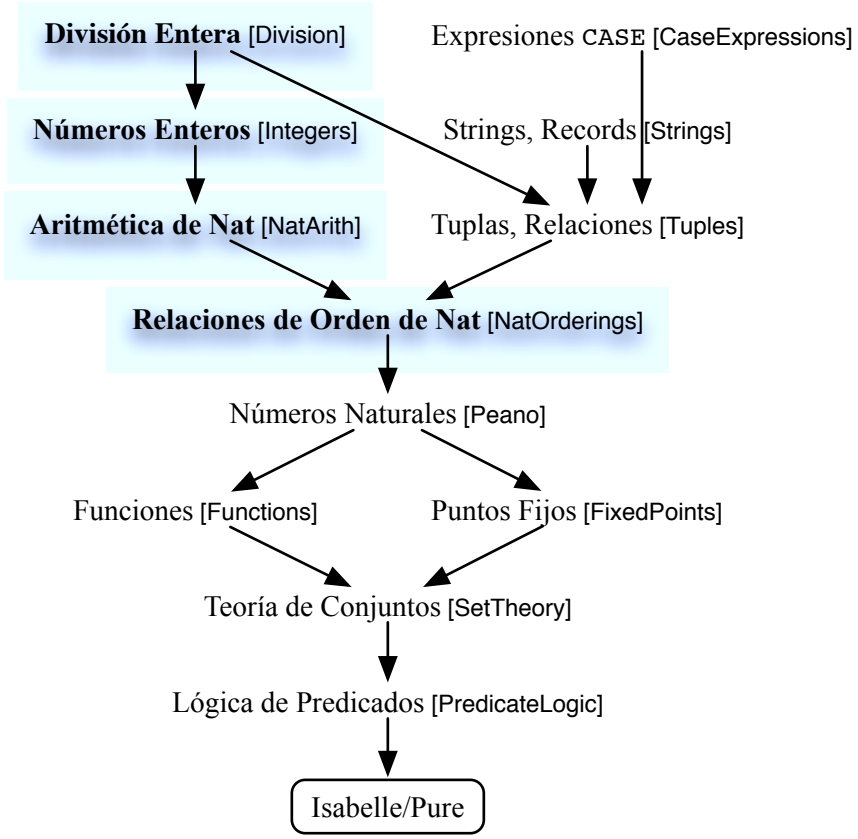

Figura 4.1: Dependencias entre las teorías de Isabelle/TLA+

La lógica proposicional se presenta de una forma no estándar. Los valores de verdad TRUE y FALSE, y las expresiones de condición (IF _ THEN _ ELSE _) se declaran como constantes. Los conectivos Booleanos se definen de manera que devuelvan solo los valores TRUE o FALSE, más allá de cuáles sean sus argumentos. Esto permite probar varias leyes de la lógica proposicional, lo que es útil para el razonamiento automático basado en reescritura.

Debido a que TLA+ es no-tipado, la equivalencia es diferente a la igualdad. Se debe tener cuidado al expresar leyes de la lógica proposicional. Por ejemplo, aunque la equivalencia $\text{TRUE} \wedge A \Leftrightarrow A$ es válida, no se puede afirmar la ley $\text{TRUE} \wedge A = A$, ya que no hay forma de conocer el valor de $\text{TRUE} \wedge 3$, por ejemplo. Estas igualdades son válidas sólo si son aplicadas a expresiones Booleanas. Para que puedan ser usadas por los métodos automáticos, se introducen el predicado auxiliar *isBool*, que es verdadero precisamente cuando los argumentos son Booleanos, y la operación *boolify*, que convierte argumentos arbitrarios a una expresión Booleana equivalente (en el sentido de $\Leftrightarrow$). Por lo tanto, para cada predicado u operador Booleano P definido en las teorías, deben probarse los lemas:

lemma *[simp]: "boolify(P(x)) = P(x)"*
lemma *[intro!, simp]: "isBool(P(x))"*

Lógica de primer orden

CHOOSE $x : p$
$\forall x : p \qquad \exists x : p$

El operador CHOOSE ya se explicó en la sección 2.2. Aquí se lo define axiomáticamente y de este se deriva la definición de los cuantificadores "para todo" y "existe".

4.2. Teoría de conjuntos

Esta teoría define la versión de la teoría de conjuntos tradicional Zermelo-Fränkel que subyace a TLA+. Se define la sintaxis básica y la axiomatización de la teoría de conjuntos. Aquí se definen las versiones ligadas de los cuantificadores y del operador CHOOSE.

En esta teoría se definen los siguientes símbolos y operadores estándar:

BOOLEAN [el conjunto {TRUE, FALSE }]
$\in \quad \notin \quad \cup \quad \cap \quad \subseteq \quad \setminus$ [diferencia de conjuntos]
$\{e_1, \ldots, e_n\}$ [El conjunto de elementos e_i]
$\{x \in S : p\}$ [El conjunto de elementos e_i de S que satisfacen p]
$\{e : x \in S\}$ [El conjunto de elementos e tal que x pertenece a S]
SUBSET S [El conjunto de subconjuntos de S]
UNION S [Unión de todos los elementos de S]
$\forall x \in S : p \qquad \exists x \in S : p \qquad$ CHOOSE $x \in S : p$

4.3. Funciones

En TLA+ las funciones no son definidas (por ejemplo, como un conjunto de pares), sino que son axiomatizadas. Es decir, se declara un conjunto de axiomas a partir del cual se define la sintaxis de las funciones y se demuestran sus propiedades. De hecho, los pares y tuplas son definidos más adelante como funciones especiales. Justamente, este enfoque ayuda a identificar las funciones y a automatizar su razonamiento (con el predicado *IsAFcn*, definido a partir de uno de los axiomas).

En esta teoría se definen los siguientes símbolos y expresiones de la sintaxis estándar:

$f[e]$	[Aplicación de función]
DOMAIN f	[Dominio de la función f]
$[x \in S \mapsto e]$	[Función f tal que $f[x] = e$ para $x \in S$]
$[S \to T]$	[Conjunto de funciones f con $f[x] \in T$ para $x \in S$]
$[f$ EXCEPT $![e_1] = e_2]$	[Función $\hat{f}$ igual a f excepto que $\hat{f}[e_1] = e_2$. Un @ en lugar de e_2 equivale a $f[e_1]$.]

Se consideran sólo funciones unarias; las funciones con múltiples argumentos son definidas como funciones sobre productos. También se definen otros operadores de función como imagen de un conjunto bajo una función y rango de una función, y los predicados de inyectividad, biyectividad y de función inversa.

4.4. Puntos fijos

En esta teoría se desarrolla el teorema de Knaster-Tarski, que establece que cualquier función monótona en un látice completo tiene un mínimo punto fijo (least fixed point, lfp) y un máximo punto fijo (gfp). Estos conceptos son usados luego en la teoría Peano (en la sección 4.5), en la construcción de los números naturales, donde se prueba la existencia de una estructura que satisface los axiomas de Peano, y en la definición recursiva de los conjuntos *upto* ($upto(n) = \{0, \ldots, n\}$). También sirven como test de la formalización de la teoría de conjuntos.

4.5. Números naturales

Los axiomas de Peano son usados como el formalismo estándar de los números naturales en la lógica matemática. A partir de este conjunto de cinco axiomas se definen los elementos que formarán el conjunto de números naturales. Para esto, se prueba la existencia de un predicado sobre los argumentos *N*, *Z* y *Succ* que satisfaga estos axiomas y de este se derivan los respectivos elementos *Nat*, *Zero*, SUCC.

La representación tradicional de los números naturales se da explícitamente para cada uno como una constante que se traduce en una sucesión de aplicaciones de SUCC. Por ejemplo:

syntax
"1" :: "c" ("1") — Declaración de la constante ''1''.
translations
"1" ⇌ *"Succ[0]"*

Además de las reglas básicas de introducción y eliminación de los elementos definidos, se define el principio de inducción para naturales (la regla *natInductE*) y la regla para el análisis de casos de naturales (*natCases*). También se incluyen los intervalos de números naturales iniciales, es decir, los intervalos de 1 hasta *n* incluido (1 .. *n*). Se definen inductivamente como el conjunto más chico de números naturales que contiene a *n* y es cerrado bajo el predecesor.

La relación de orden entre naturales "mayor o igual" (≤) estaba definida en esta teoría en función del operador upto. En este trabajo, como se verá más adelante, se extendió la definición a la relación de orden estricta (<). La gran cantidad de propiedades declaradas y probadas llevó a mover todos estos conceptos a la nueva teoría *NatOrderings*.

4.6. Tuplas, Relaciones, Strings y Records

La formalización de Isabelle/TLA+ se ha continuado desarrollando más allá de la aritmética de los números enteros que se ha hecho en este trabajo. Es así que a partir de éste se fueron agregando los restantes operadores constantes de TLA+ que dependen de los conceptos de relaciones de orden de naturales y de los operadores aritméticos.

Tuplas y Relaciones Las tuplas de tamaño *n* son funciones cuyos dominios son los intervalos de la forma 1 .. *n*, es decir, la secuencia de números naturales de 1 a *n*. Las relaciones son conjuntos de tuplas. En esta teoría también se introduce la notación $[x : S, y : T]$, que denota el conjunto de funciones f con dominio $\{x, y\}$ tal que $f[x] \in S$ y $f[y] \in T$, y la nociones estándar de relaciones binarias, como relaciones de orden y relaciones de equivalencia. También se formalizan las expresiones CASE.

Strings Los caracteres son representados como pares de números hexadecimales (de hecho, números naturales del 0 al 15) y los strings como secuencias de caracteres.

Records Son representados como funciones con dominio en Strings. Por ejemplo, $(''foo'' :> 1)@@(''bar'' :>$ TRUE). Los conjuntos de records se representan como conjuntos de funciones enumeradas, por ejemplo: $[''foo'' : Nat, ''bar'' :$ BOOLEAN]. La implementación de la sintaxis estándar de TLA+ para records no es sencilla, ya que el lexer de Isabelle distingue entre identificadores y strings: estos últimos deben aparecer entre dos comillas simples.

Capítulo 5

Formalización de la aritmética de los números naturales

En este capítulo describiremos uno de los principales aportes de este trabajo: la formalización de las conceptos aritméticos del lenguaje TLA$^+$ en el lógica-objeto Isabelle/TLA$^+$. Definiremos los operadores aritméticos dados en los módulos estándar de TLA$^+$ *Naturals* e *Integers* para que satisfagan la sintaxis y semántica de TLA$^+$ allí definidas (ver sección 2.2.3). Estos operadores son: suma, multiplicación, exponenciación, resta, división y módulo. En este capítulo daremos sus definiciones con argumentos pertenecientes al conjunto de los números naturales y en el siguiente capítulo, con argumentos enteros.

La mayoría de las funciones estudiadas en teoría de números, como por ejemplo suma, división, factorial, exponencial, etc., y las que queremos definir aquí que son un subconjunto de estas, son *recursivas primitivas*. De hecho es difícil encontrar funciones que no lo sean. Para definirlas recursivamente en Isabelle/TLA$^+$ necesitamos un *esquema de recursión primitiva* sobre los números naturales, que lo declararemos como un nuevo axioma. Con el esquema de recursión podemos definir las funciones aritméticas y probar sus propiedades algebraicas.

El conjunto de reglas aquí declaradas está basado en las reglas dadas en las lógicas Isabelle/HOL e Isabelle/ZF. Aunque son sustancialmente distintas, se extrajo información sobre qué teoremas agregar al Simplificador o al Razonador Clásico.

Por falta de espacio, sólo presentaremos las pruebas de los teoremas que consideremos relevantes. Todos los teoremas o lemas aquí declarados están probados; las pruebas que no estén presentes aquí se encontrarán en el apéndice A.

5.1. Esquema de recursión sobre los naturales

Declaramos como axioma un esquema para definir funciones recursivas primitivas con un argumento y dominio en los naturales, que luego será usado para definir la suma, la multiplicación, la resta y la exponenciación entre naturales.

axioms

primrec_nat: “$\exists f : isAFcn(f) \wedge DOMAIN\ f = Nat$
$\wedge\ f[0] = e$
$\wedge\ (\forall n \in Nat : f[\text{Succ}[n]] = h(n, f[n]))$”

El axioma *primrec_nat* nos asegura que existe f, que es una función con dominio en *Nat* y que f tiene un caso base y un paso recursivo. La aplicación de f en el caso base (0) devuelve la expresión e; la aplicación de f en $n+1$ devuelve una función h que depende de n y de la aplicación de f en el paso

anterior, $f[n]$. Las variables no ligadas e y h son variables esquemáticas de Isabelle (ver sección 3.3.1), es decir que pueden instanciarse con cualquier expresión que cumplan respectivamente con $e \in S$ y $\forall n \in Nat, x \in S : h(n,x) \in S$, donde la función $f \in [Nat \to S]$ ($f \in$ 'conjunto de funciones de Nat en S').
Por lo tanto el axioma *primrec_nat* en realidad es un *esquema de axiomas*: permite instanciar infinitos teoremas a partir de este, correspondientes a infinitas funciones recursivas primitivas.

Esta es sólo una de las formas de definir un esquema de recursión sobre los naturales. Se puede comprobar la necesidad de la fórmula que representa este axioma al intentar probar teoremas básicos de funciones primitivas, como haremos más adelante. En las pruebas se llega a un punto donde uno de los subobjetivos es la fórmula expresada por *primrec_nat* y no hay manera de probarla a partir de las teorías anteriormente definidas. Simplemente por este motivo podemos justificar la declaración de *primrec_nat* como axioma.

Del axioma anterior derivamos fácilmente el lema equivalente *bprimrec_nat*. Este lema, en estructura similar al axioma, difiere en que la variable f está ligada ($f \in [Nat \to S]$) y donde las condiciones sobre las variables libres están como premisas. Esto permitirá más adelante instanciarlas para facilitar las pruebas. Este lema se prueba fácilmente del axioma.

lemma *bprimrec_nat:*
assumes *"e ∈ S"* **and** *"∀ n ∈ Nat : ∀ x ∈ S : h(n,x) ∈ S"*
shows *"∃f ∈ [Nat → S] : f[0] = e ∧ (∀ n ∈ Nat: f[Succ[n]] = h(n,f[n]))"*

El axioma y el lema que acabamos de definir se agregan como parte de la teoría *Peano*.

5.2. Suma

Intuitivamente, podemos definir de manera recursiva a la función suma f con las reglas

$$\begin{aligned} f(m,0) &= m \\ f(m,n+1) &= f(m,n)+1 \end{aligned}$$

que, en la notación de TLA$^+$, son equivalentes a las ecuaciones

$$m + 0 = m \tag{5.1}$$
$$m + \text{Succ}[n] = \text{Succ}[m+n] \tag{5.2}$$

El argumento sobre el que aplicamos recursión es el segundo. Vemos que el resultado del caso base es m y el paso recursivo es aplicar la función sucesor a Succ$[m+n]$. En el esquema del lema *bprimrec_nat*, podemos instanciar las variables libres satisfaciendo las hipótesis del lema original:

$$\begin{aligned} e &\equiv m \\ S &\equiv Nat \\ h &\equiv \lambda\, n\ f.\ \text{Succ}[f] \end{aligned}$$

Así obtenemos el nuevo lema en Isabelle *bprimrec_nat*[of m Nat "$\lambda\, n f$.Succ[f]"] que equivale a la expresión:

$$\begin{aligned} &[\![m \in Nat; \forall n, x \in Nat : Succ[x] \in Nat]\!] \\ &\quad \Longrightarrow \exists\, f \in [Nat \to Nat] : f[0] = m \wedge (\forall\, n \in Nat : f[\text{Succ}[n]] = \text{Succ}[f[n]]) \end{aligned} \tag{5.3}$$

que demuestra que existe la función recursiva que queremos definir, si se cumplen las premisas. Para construirla debemos aplicar esta función al argumento n. La función recursiva de la suma, llamada *primrec_add*(m,n), la definimos con el operador CHOOSE de la siguiente manera:

definition *primrec_add* **where**
"primrec_add(m,n) ≡ LET g ≡ (CHOOSE f ∈ [Nat → Nat] :
f[0] = m
∧ (∀ x ∈ Nat : f[Succ[x]] = Succ[f[x]]))
IN g[n]"

Una de las premisas de *bprimrec_nat* es $e \in Nat$, que en el lema 5.3 equivale a $m \in Nat$. Como f tiene dominio en Nat es necesario que la variable $n \in Nat$. Ya que en Isabelle/TLA$^+$ no hay inferencia de tipos, por seguridad definimos una nueva función *addn* que llama a la recién definida *primrec_add*, y a la que se le agregan como condición que ambos argumentos pertenezcan al conjunto *Nat*. A raíz de esto, es que todos los lemas que usen esta definición también deben incluir como hipótesis que los argumentos pertenezcan al conjunto de los naturales, para poder "desarmar" la definición de suma. Al mismo tiempo, además le damos la notación usual de la suma con el comando **infixl**, donde el número 65 es el nivel de precedencia del operador.

definition *addn :: " [c,c] ⇒ c"* (**infixl** *"+" 65*)
where *nat_add_def: "⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ (m + n) ≡ primrec_add(m,n)"*

5.2.1. La propiedad de clausura y su prueba

Ya tenemos las definiciones, ahora comenzaremos a probar propiedades sobre ellas. En primer lugar probamos la clausura de la suma. Desarrollaremos esta primer prueba en pasos sencillos y mostrando los subobjetivos intermedios.

lemma *addIsNat [intro!,simp]:*
"⟦ m ∈ Nat ; n ∈ Nat ⟧ ⟹ m + n ∈ Nat"
unfolding *nat_add_def primrec_add_def Let_def*

Este lema será muy útil en muchas de las pruebas de lemas que incluyan suma. Con el atributo *simp*, estamos agregando el lema al Simplificador y con el tributo *intro*! estamos declarando al lema como una regla de introducción segura, información que será usado por el *razonador clásico*. (Sobre los atributos, ver sección 3.2.)

Para probarlo, comenzamos desarmando todas las definiciones sintácticas con el comando **unfolding** hasta un nivel que permita aplicar algún lema ya conocido. Ahora, el nuevo objetivo es:

1. ⟦ $m \in Nat$; $n \in Nat$ ⟧ ⟹ (CHOOSE $f \in [Nat \to Nat]$:
   $f[0] = m \wedge (\forall x \in Nat : f[Succ[x]] = Succ[f[x]]))[n] \in Nat$

Aquí podemos usar la regla de introducción del operador CHOOSE llamada *bChooseI2*, definida en la teoría *PredicateLogic* para deshacernos del operador.

$$\frac{\exists x \in ?A : ?P(x) \qquad \bigwedge x.[\![x \in ?A; ?P(x)]\!] \Longrightarrow ?Q(x)}{?Q(\text{CHOOSE } x \in ?A : ?P(x))} \quad bChooseI2$$

Aplicamos la regla usando el método *rule* (ver 3.3.1), que intenta *unificar* el objetivo con la conclusión de la regla:

apply *(rule bChooseI2)*

La aplicación del método es exitoso y dos nuevos subobjetivos son derivados, correspondientes a las dos premisas de la regla aplicada:

1. $[\![\ m \in Nat;\ n \in Nat\ ]\!]$
   $\Longrightarrow \exists x \in [Nat \to Nat] : x[0] = m \wedge (\forall xa \in Nat : x[\text{Succ}[xa]] = \text{Succ}[x[xa]])$
2. $\bigwedge x.\ [\![\ m \in Nat;\ n \in Nat;\ x \in [Nat \to Nat];$
   $x[0] = m \wedge (\forall xa \in Nat : x[\text{Succ}[xa]] = \text{Succ}[x[xa]])\ ]\!]$
   $\Longrightarrow x[n] \in Nat$

El primer subobjetivo es exactamente el lema 5.3 que derivamos de *bprimrec_nat*. Aquí notamos que no es posible demostrar este subobjetivo a partir de los teoremas ya definidos, excepto si usamos el esquema de recursión. El segundo subobjetivo es una propiedad de las funciones, que se puede probar usando sólo las premisas $x \in [Nat \to Nat]$ y $n \in Nat$.

Entonces usamos el lema *bprimrec_nat* sobre el objetivo 1, otra vez por unificación:

apply *(rule bprimrec_nat)*

El siguiente paso es probar las hipótesis derivadas del lema recién aplicado, que son los nuevos subojetivos 1 y 2:

1. $[\![\ m \in Nat;\ n \in Nat\ ]\!] \Longrightarrow m \in Nat$
2. $[\![\ m \in Nat;\ n \in Nat\ ]\!] \Longrightarrow \forall x \in Nat : \text{Succ}[x] \in Nat$
3. $\bigwedge x.\ [\![\ m \in Nat;\ n \in Nat;\ x \in [Nat \to Nat];$
   $x[0] = m \wedge (\forall xa \in Nat : x[\text{Succ}[xa]] = \text{Succ}[x[xa]])\ ]\!]$
   $\Longrightarrow x[n] \in Nat$

Se puede finalizar la prueba con la táctica automática *auto*. La prueba completa del lema puede resumirse en estos dos pasos:

unfolding *nat_add_def primrec_add_def Let_def*
by *(rule bChooseI2, rule bprimrec_nat, auto)*

5.2.2. El caso base y el paso recursivo

Las siguientes propiedades de la suma que probaremos son el caso base y el paso recursivo de la definición, esto es, respectivamente, las fórmulas 5.1 y 5.2. Estos lemas, por ser justamente las dos ecuaciones que definen la función suma, serán usados de aquí en adelante en lugar de la definición explícita del operador cuando se deba referirse a ella en alguna prueba.

lemma *add_0_nat [simp]:* — Caso base 5.1
"n ∈ Nat ⟹ n + 0 = n"
apply *(simp add: nat_add_def primrec_add_def Let_def)*
by *(rule bChooseI2, rule bprimrec_nat, auto)*

lemma *add_Succ_nat [simp]:* — Paso recursivo 5.2
"[\| m ∈ Nat; n ∈ Nat \|] ⟹ m + Succ[n] = Succ[m + n]"
apply *(simp add: nat_add_def primrec_add_def Let_def)*
by *(rule bChooseI2, rule bprimrec_nat, auto)*

Las pruebas son bastantes similares a la prueba anterior. Esta vez el comando **unfolding** no bastará para desarmar las definiciones, ya que faltan como hipótesis que ambos argumentos pertenezcan a *Nat*. Es por esto que usamos el Simplificador que ya cuenta implícitamente con las reglas *zeroIsNat*: "0 ∈ *Nat*", usada en el primer caso, y *succIsNat*: "$x \in Nat \Longrightarrow \text{Succ}[x] \in Nat$", en el segundo. El resto de las pruebas se completa en ambos casos de la misma forma que en la prueba de *addIsNat*.

De ahora en más, para cada operador definido en base al esquema de recursión para los naturales, las pruebas de la propiedad de clausura y de los lemas del caso base y del paso recursivo serán prácticamente las mismas (obviamente cambiando el lema de definición del operador en cuestión). Estos operadores son: suma, multiplicación, resta y exponenciación.

5.2.3. Otras propiedades algebraicas

A partir de ahora, no necesitaremos usar el axioma de esquema de recursión en las pruebas sobre la suma. Sólo bastarán, en principio, los tres lemas que acabamos de definir y que hemos agregado al Simplificador. Tampoco será necesario desde ahora desarmar la definición de suma, todo se irá construyendo a partir de estos tres lemas base.

La propiedad que queremos probar ahora es $0 + n = n$:

lemma *add_0_left_nat [simp]: "n ∈ Nat ⟹ 0 + n = n"*

A diferencia del lema anterior *add_0_nat* (simétrico a este), el argumento recursivo no corresponde a ninguno de los dos del esquema recursivo. Si comenzamos la prueba desarmando las definiciones, llegamos a un punto en donde no podemos avanzar con la prueba. En lugar de esto, aplicaremos el método inductivo sobre la única variable. El método inductivo, aplicado a los naturales, usa un *principio de inducción* definido por la regla *natInductE* definida en la teoría *Peano*:

$$\frac{n \in Nat \qquad P(0) \qquad \bigwedge x.[\![x \in Nat; P(x)]\!] \Longrightarrow P(\text{Succ}[x])}{P(n)} \quad natInductE$$

La aplicación del principio de inducción de los naturales genera dos nuevos subobjetivos: el caso base y el paso inductivo.

apply *(induct set: Nat)*

1. $0 + 0 = 0$
2. $\bigwedge x.\ [\![\ x \in Nat;\ 0 + x = x\ ]\!] \Longrightarrow 0 + \text{Succ}[x] = \text{Succ}[x]$

Podemos terminar la prueba usando el Simplificador en ambos subobjetivos. En el primer caso, el Simplificador aplicará implícitamente el lema *add_0_nat*. En el segundo, aplicará *add_Succ_nat* para reescribir el objetivo, primero en "Succ[0 + x] = Succ[x]", y luego en "Succ[x] = Succ[x]", usando la hipótesis inductiva. La prueba completa es:

by *(induct set: Nat, simp_all)*

El lema simétrico de *add_Succ_nat* es *add_Succ_left_nat* y se prueba también por inducción[1], como la mayoría de las propiedades de la suma que siguen.

lemma *add_Succ_left_nat [simp]:*
"[[n ∈ Nat; m ∈ Nat]] ⟹ Succ[m] + n = Succ[m + n]" **by** *(induct set: Nat, simp_all)*

La propiedad conmutativa y la propiedad asociativa no se pueden agregar al Simplificador.

lemma *add_commute_nat:* — Propiedad conmutativa de la suma.
"[[m ∈ Nat; n ∈ Nat]] ⟹ m + n = n + m" **by** *(induct set: Nat, simp_all)*
lemma *add_assoc_nat:* — Propiedad asociativa de la suma.
"[[m ∈ Nat; n ∈ Nat; p ∈ Nat]] ⟹ m + (n + p) = (m + n) + p" **by** *(induct set: Nat, simp_all)*

[1] En este caso el orden de los argumentos está cambiado para que se aplique el método de inducción sobre el primero de ellos. Esto no afecta en nada a las pruebas subsiguientes.

Ninguno de los lados de la ecuación es estructuralmente más simple que el otro. La reescritura de términos con estas reglas lleva al Simplificador a un bucle del que no puede salir. Es por esto que habrá que usarlas cuidadosamente y dependiendo de cada caso. La propiedad cancelativa, en cambio, es apropiada como regla de reescritura ya que se deshace de una de las variables.

lemma *add_left_cancel_nat [simp]:* — Propiedad cancelativa por izquierda.
"⟦ *m* ∈ *Nat*; *n* ∈ *Nat*; *p* ∈ *Nat* ⟧ ⟹ *(m + n = m + p) = (n = p)*" **by** *(induct set: Nat, simp_all)*
lemma *add_right_cancel_nat [simp]:* — Propiedad cancelativa por derecha.
"⟦ *m* ∈ *Nat*; *n* ∈ *Nat*; *p* ∈ *Nat* ⟧ ⟹ *(n + m = p + m) = (n = p)*" **by** *(simp add: add_commute_nat)*

La segunda regla cancelativa se prueba usando implícitamente la primer regla cancelativa (con el simplificador) y la regla conmutativa de la suma. En general, todas las reglas en las que haya sumas (y/o multiplicaciones, más adelante) se declararán en varias versiones. De ahora en más, las versiones de los lemas que se prueben agregando las propiedades conmutativas (como *add_right_cancel_nat*) no las mostraremos.

Otras de las propiedades que se probaron por inducción son las siguientes:

lemma *add_is_0_nat [simp]:*
"⟦ *m* ∈ *Nat*; *n* ∈ *Nat* ⟧ ⟹ *(m + n = 0) = (m = 0 ∧ n = 0)*"
lemma *add_is_1_nat:*
"⟦ *m* ∈ *Nat*; *n* ∈ *Nat* ⟧ ⟹ *(m + n = 1) = (m = 1 ∧ n = 0 ∨ m = 0 ∧ n = 1)*"
lemma *add_eq_self_zero_nat [simp]:*
"⟦ *m* ∈ *Nat*; *n* ∈ *Nat* ⟧ ⟹ *(m + n = m) = (n = 0)*"
lemma *add_left_commute_nat:*
"⟦ *m* ∈ *Nat*; *n* ∈ *Nat*; *p* ∈ *Nat* ⟧ ⟹ *m + (n + p) = n + (m + p)*"

theorems *add_ac_nat = add_assoc_nat add_commute_nat add_left_commute_nat*

En *add_ac_nat* juntamos los lemas sobre asociatividad y conmutatividad que, en general, son usan de manera conjunta en las pruebas (como haremos más adelante en *mult_Succ_left_nat*).

5.3. Multiplicación

La multiplicación sobre los números naturales se define recursivamente en el segundo argumento, de acuerdo a las siguientes ecuaciones:

$$m * 0 = 0 \tag{5.4}$$
$$m * \text{Succ}[n] = m * n + m \tag{5.5}$$

De la misma manera que para la suma, usamos el operador CHOOSE y el esquema de recursión primitiva para definir el operador de multiplicación en Isabelle/TLA$^+$, con nombre *multnat*. Luego damos la definición de la función que "envuelve" a la anterior (*nat_mult_def*) y que tiene como premisas que los argumentos pertenezcan al conjunto de los naturales, junto con la notación estándar de la multiplicación.

definition *primrec_mult* **where**
"*primrec_mult(m,n)* ≡ LET *f* ≡ (CHOOSE *g* ∈ *[Nat → Nat] :*
g[0] = 0
∧ (∀ x ∈ Nat : g[Succ[x]] = g[x] + m))
IN *f[n]*"

definition *mult :: "[c,c] ⇒ c" (***infixl** *"∗" 70)*
where *nat_mult_def: "⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ (m ∗ n) ≡ primrec_mult(m,n)"*

A partir de esta definición y el axioma del esquema de recursión de los naturales *bprimrec_nat*, probamos los tres lemas base sobre el que construiremos las demás propiedades: la clausura de la multiplicación (*multIsNat*), el caso base (*mult_0_nat*) y el paso recursivo (*mult_Succ_nat*). Las pruebas son básicamente las mismas que las de los lemas base de la suma.

lemma *multIsNat [intro!, simp]: "⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ m ∗ n ∈ Nat"*
lemma *mult_0_nat [simp]: "n ∈ Nat ⟹ n ∗ 0 = 0"*
lemma *mult_Succ_nat [simp]: "⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ m ∗ Succ[n] = m ∗ n + m"*

Los siguientes lemas, simétricos del caso base y el paso recursivo respectivamente, se prueban por inducción, al igual que los de la suma.

lemma *mult_0_left_nat [simp]: "n ∈ Nat ⟹ 0 ∗ n = 0"*
by *(induct set: Nat, simp_all)*
lemma *mult_Succ_left_nat [simp]: "⟦ n ∈ Nat; m ∈ Nat ⟧⟹ Succ[m] ∗ n = m ∗ n + n"*
by *(induct set: Nat, simp_all add: add_ac_nat)*

También se probaron por inducción las propiedades conmutativa, distributiva por izquierda y por derecha entre la suma y la multiplicación, y asociativa.

lemma *mult_commute_nat:* — Propiedad conmutativa de la multiplicación.
"⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ m ∗ n = n ∗ m"
by *(induct set: Nat, simp_all)*

lemma *add_mult_distrib_left_nat:* — Propiedad distributiva por izquierda de + y ∗.
"⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat ⟧ ⟹ m ∗ (n + p) = m ∗ n + m ∗ p"
proof *(induct set: Nat, simp_all)*
fix *m* **assume** *nat: "m ∈ Nat" "n ∈ Nat" "p ∈ Nat"* **and** *"m ∗ (n + p) = m ∗ n + m ∗ p"*
hence *"m ∗ (n + p) + (n + p) = m ∗ n + m ∗ p + (n + p)"*
by *(simp only: add_right_cancel_nat)*
hence *"m ∗ n + m ∗ p + (n + p) = m ∗ n + (m ∗ p + n) + p"*
using *nat* **by** *(simp add: add_assoc_nat)*
also have *"... = m ∗ n + (n + m ∗ p) + p"*
using *nat* **by** *(simp add: add_commute_nat)*
finally show *"m ∗ n + m ∗ p + (n + p) = m ∗ n + n + (m ∗ p + p)"*
using *nat* **by** *(simp add: add_assoc_nat)*
qed

lemma *mult_assoc_nat:* — Propiedad asociativa de la multiplicación.
"⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ m ∗ (n ∗ p) = (m ∗ n) ∗ p"
by *(induct set: Nat, auto simp add: add_mult_distrib_right_nat)*

El caso base es trivial; la prueba en estilo Isar corresponde al paso inductivo. Como puede verse, las propiedades conmutativas y asociativas de la suma son usadas en determinados pasos intermedios de la prueba. Las pruebas de las dos propiedades distributivas no son triviales para los métodos automáticos. Lo ideal sería probar el lema con un comando como:

by *(induct set: Nat, simp_all add: add_commute_nat add_assoc_nat)*

El problema es que se falla al hacer unificación entre las premisas de las reglas *add_commute_nat* y *add_assoc_nat* y expresiones como $(n + p) \in Nat$ o $(m * n) \in Nat$. La segunda prueba de distributividad sí se prueba automáticamente usando la propiedad distributiva inmediatemente anterior probada y por conmutatividad de la suma.

Al momento de entrar a la prueba en estilo Isar, el subobjetivo actual puede verse directamente en la prueba. La variable *m* (distinta de la anterior) es una meta-variable, que se fija con el comando **fix**. Las hipótesis del subobjetivo son las fórmulas que aparecen después de **assume**. La conclusión del subobjetivo a probar es la fórmula que aparece después del comando **show**. En el medio puede verse el desarrollo de los pasos intermedios con sus pruebas hasta llegar a la conclusión.

Se agregaron también las siguientes propiedades al Simplificador, que se prueban por inducción.

lemma *mult_is_0_nat [simp]: "⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ (m * n = 0) = (m = 0 ∨ n = 0)"*
lemma *mult_eq_1_iff_nat [simp]: "⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ (m * n = 1) = (m = 1 ∧ n = 1)"*

Otra propiedad cuya prueba no es trivial es la cancelativa de la multiplicación. La dificultad está en que si la variable cancelada *k* es 0, las otras variables *m* y *n* no necesariamente son iguales. A continuación mostramos una prueba posible [2].

lemma *mult_cancel1_nat [simp]:*
assumes *k: "k ∈ Nat"* **and** *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"(k * m = k * n) = (m = n ∨ (k = 0))"*
using *n m k* **apply** *(induct arbitrary: m, auto)*
apply *(rule_tac n=ma in natCases, simp_all)*
by *(force elim!: natCases)*

Finalmente, y de la misma manera que para la suma, agrupamos los lemas no agregados al Simplificador en *mult_ac_nat*.

theorems *mult_ac_nat = mult_assoc_nat mult_commute_nat mult_left_commute_nat*

5.4. Relaciones de Orden

La resta entre naturales, $x - y$ donde $x, y \in \mathbb{N}$ tiene la particularidad que si $y > x$ entonces $x - y = 0$. Entonces, para definir este operador, antes necesitamos desarrollar la teoría de las relaciones de orden entre naturales.

Partimos de la base que la noción intuitiva de orden entre número naturales ($\leq$) ya está definida como parte de la teoría *Peano*. La gran cantidad de lemas añadidos llevó a crear una nueva teoría, a la que llamamos *NatOrderings*.

La constante $<$ la definimos sobre cualquier dominio de la siguiente manera:

definition *less :: "[c,c] ⇒ c"* *(***infixl** *"<" 50)*
where *"a < b ≡ a ≤ b ∧ a ≠ b"*

La relación $\geq$ es simplemente la abreviación sintáctica $x \geq y \equiv y \leq x$, al igual que la relación $>$ es $x > y \equiv y < x$.

Hay varias formas de expresar la irreflexividad de $<$. Una es como en el lema *less_not_refl*. Los lemas *less_irrefl* y *less_irreflE* expresan lo mismo pero de un forma conveniente para los métodos automáticos. La transitividad se prueba en el lema *leq_ neq_trans* en forma de regla de eliminación.

[2] En el apéndice, esta misma prueba está desarrollada en Isar; es mucho más larga (son 42 pasos) pero también más legible.

lemma *less_not_refl: "a < b ⟹ a ≠ b"* **by** *auto*
lemma *less_irrefl [simp]: "(a < a) = FALSE"* **by** *(simp add: less_def)*
lemma *less_irreflE [elim!]: "a < a ⟹ R"* **by** *simp*
lemma *less_imp_leq [elim]: "a < b ⟹ a ≤ b"* **by** *(simp add: less_def)*
lemma *leq_neq_trans [elim]: "a ≤ b ⟹ a ≠ b ⟹ a < b"* **by** *(simp add: less_def)*
lemma *leq_neq_iff_less: "a ≤ b ⟹ (a ≠ b) = (a < b)"* **by** *auto*

Hay muchos lemas que se pueden probar sobre las relaciones de orden. Sin embargo, hay que tener cuidado al seleccionar cuáles de ellos agregar a las tácticas de prueba automáticas. Por ejemplo, el lema *leq_neq_iff_less* parece lógico agregarlo al Simplificador. Si $a \leq b$ es una hipótesis, reemplazar $a \neq b$ por $a < b$. Pero el Simplificador intentará unificar todas las desigualdades presentes en el estado de prueba, lo que puede llevar mucho tiempo o incluso a la no terminación.

Aquí se mostrarán sólo las declaraciones de los lemas principales. Sus pruebas en general son de un paso, usando algún lema anterior y/o aplicando *natCases*. La lista completa junto con las pruebas puede consultarse en el apéndice A.

lemma *nat_Succ_leq_iff_less [simp]:* — Definición alternativa de <.
"⟦ m ∈ Nat; n ∈ Nat⟧ ⟹ (Succ[m] ≤ n) = (m < n)"
by *(auto simp add: less_def dest: nat_Succ_leqD nat_leq_limit)*

lemma *nat_leq_less:* — Reducción de ≤ a <.
"⟦ m ∈ Nat; n ∈ Nat⟧ ⟹ m ≤ n = (m < n ∨ m = n)"
by *(auto simp add: less_def)*

lemma *nat_less_Succ_iff_leq [simp]:*
"⟦ m ∈ Nat; n ∈ Nat⟧ ⟹ (m < Succ[n]) = (m ≤ n)"
by *(simp del: nat_Succ_leq_iff_less add: nat_less_iff_Succ_leq nat_Succ_leq_Succ)*

Sobre < y *Succ*, probados con auto y *less_def*

lemma *nat_not_less0 [simp]: "n ∈ Nat ⟹ (n < 0) = FALSE"*
lemma *nat_less_SuccI: "⟦ m < n; m ∈ Nat; n ∈ Nat⟧ ⟹ m < Succ[n]"*
lemma *nat_Succ_lessD: "⟦ Succ[m] < n; m ∈ Nat; n ∈ Nat⟧ ⟹ m < n"*
lemma *nat_less_leq_not_leq: "⟦ m ∈ Nat; n ∈ Nat⟧ ⟹ (m < n) = (m ≤ n ∧ ¬ n ≤ m)"*
lemma *nat_Succ_not_less_self [simp]: "n ∈ Nat ⟹ Succ[n] < n = FALSE"*

Sobre transitividad de <, probada en base a la transitividad de ≤.

lemma *nat_less_trans: "⟦ k < m; m < n; k ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ k < n"*
lemma *nat_less_trans_Succ: "⟦ i < j; j < n; i ∈ Nat; j ∈ Nat; k ∈ Nat⟧ ⟹ Succ[i] < k"*
lemma *nat_leq_less_trans: "⟦ k ≤ m; m < n; k ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ k < n"*
lemma *nat_less_leq_trans: "⟦ k < m; m ≤ n; k ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ k < n"*

Sobre linearidad. La relación ≤ es lineal o total si $a \geq b \vee b \leq a$, e implica reflexividad ($a \leq a$).

lemma *nat_less_linear:* — < es lineal.
"⟦ m ∈ Nat; n ∈ Nat⟧ ⟹ m < n ∨ m = n ∨ n < m"
lemma *nat_leq_less_linear:* — Linearidad de ≤ y <.
"⟦ m ∈ Nat; n ∈ Nat⟧ ⟹ m ≤ n ∨ n < m"

lemma *nat_less_cases [case_names less equal greater]:* — Análisis por casos de linearidad.
"⟦ *m* ∈ *Nat*; *n* ∈ *Nat*; (*m* < *n* ⟹ *P*); (*m* = *n* ⟹ *P*); (*n* < *m* ⟹ *P*) ⟧ ⟹ *P*"
using *nat_less_linear* **by** *blast*

lemma *nat_not_less:* "⟦ *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (¬ *m* < *n*) = (*n* ≤ *m*)"
lemma *nat_not_lessD:* "⟦ ¬ *m* < *n*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *n* ≤ *m*"
lemma *nat_not_lessI:* "⟦ *n* ≤ *m*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ ¬ *m* < *n*"
lemma *nat_not_less_iff_gr_or_eq:* "⟦ *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (¬ *m* < *n*) = (*m* > *n* ∨ *m* = *n*)"
lemma *nat_not_less_eq:* "⟦ *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (¬ *m* < *n*) = (*n* < Succ[*m*])"
lemma *nat_not_leq:* "⟦ *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (¬ *m* ≤ *n*) = (*n* < *m*)"

lemma *nat_neq_iff:* "⟦ *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *m* ≠ *n* = (*m* < *n* ∨ *n* < *m*)"

Sobre la asimetría de <.

lemma *nat_less_not_sym:* "⟦ *m* < *n*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ ¬(*n* < *m*)"
lemma *nat_antisym_conv1:* "⟦ ¬ *m* < *n*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (*m* ≤ *n*) = (*m* = *n*)"
lemma *nat_antisym_conv2:* "⟦ *m* ≤ *n*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (¬ *m* < *n*) = (*m* = *n*)"
lemma *nat_antisym_conv3:* "⟦ ¬ *n* < *m*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (¬ *m* < *n*) = (*m* = *n*)"

lemma *nat_less_antisym_false:* "⟦ *m* < *n*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *n* < *m* = *FALSE*"
lemma *nat_less_antisym_leq_false:* "⟦ *m* < *n*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *n* ≤ *m* = *FALSE*"

Sobre 0 < *n*.

lemma *nat_gt0_not0:* "*n* ∈ *Nat* ⟹ (0 < *n*) = (*n* ≠ 0)"
lemma *gt0_implies_Suc:* "⟦ *n* ∈ *Nat*; 0 < *n*⟧ ⟹ ∃ *m* ∈ *Nat* : *n* = Succ[*m*]"

5.5. Resta

Se define recursivamente la resta entre naturales a partir de las siguientes ecuaciones.

$$m \mathbin{--} 0 = m \tag{5.6}$$
$$m \mathbin{--} \text{Succ}[n] = \text{Pred}[m \mathbin{--} n] \tag{5.7}$$

Esta vez la función *h* sobre el argumento recursivo es la función predecesor Pred[3]. Definimos Pred[0] = 0, por lo tanto $m \mathbin{--} n = 0$, si $m < n$. Pred[*n*] elige un $x \in Nat$ de manera que $n = \text{Succ}[x]$, si $n \neq 0$.

definition Pred **where**
"Pred ≡ [*n* ∈ *Nat* ↦ IF *n* = 0 THEN 0 ELSE CHOOSE *x* ∈ *Nat* : *n* = Succ[*x*]]"

Usamos el símbolo −− para diferenciar esta operación de la resta estándar de TLA$^+$ entre números enteros (que usa el símbolo −). La propiedad de clausura y las ecuaciones elementales (es decir, las que definen la función) se prueban fácilmente con la definición de Pred, aplicando la regla de introducción *bChooseI2* y usando el simplificador.

[3] La función Pred no es parte de la sintaxis estándar de TLA$^+$.

lemma *PredInNat [intro!, simp]: "n ∈ Nat ⟹ Pred[n] ∈ Nat"* — Clausura.
lemma *Pred_0_nat [simp]: "Pred[0] = 0"*
lemma *Pred_Succ_nat [simp]: "n ∈ Nat ⟹ Pred[Succ[n]] = n"*
lemma *Succ_Pred_nat [simp]: "⟦ n ∈ Nat; 0 < n ⟧ ⟹ Succ[Pred[n]] = n"*

lemma *Pred_add_right_nat:* — Su prueba es por doble inducción sobre ambos argumentos.
"⟦ m ∈ Nat; n ∈ Nat; 0 < n ⟧ ⟹ Pred[m + n] = m + Pred[n]"

Ahora podemos definir la resta para naturales en el esquema de recursión:

definition *"primrec_diff"* (**infixl** *"--"*) **where**
"m -- n ≡ LET f ≡ (CHOOSE g ∈ [Nat → Nat] :
g[0] = m ∧ (∀ x ∈ Nat : g[Succ[x]] = Pred[g[x]]))
IN f[n]"

lemma *diffIsNat [intro!, simp]:* — Propiedad de clausura.
"⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ m -- n ∈ Nat"
lemma *diff_0_nat [simp]:* — Caso base.
"m ∈ Nat ⟹ m -- 0 = m"
lemma *diff_Succ_nat:* — Paso recursivo.
"⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ m -- Succ[n] = Pred[m -- n]"

El lema *diff_Succ_nat* decidimos no agregarlo por defecto al Simplificador: preferimos trabajar con el operador Succ en lugar de Pred.

También probamos casos triviales, formas de asociatividad, conmutatividad y reglas de cancelación, y distributividad con la suma y la multiplicación. Sin embargo, no se cumple la propiedad "Succ[m] $--$ n = Succ[m $--$ n]" (con $m = 0$ y $n = 1$, por ejemplo).

Los siguientes lemas son probados por inducción y por el Simplificador, agregándole la regla *diff_Succ_Nat*.

lemma *diff_0_eq_0_nat [simp]: "n ∈ Nat ⟹ 0 -- n = 0"*
lemma *diff_Succ_Succ_nat [simp]: "⟦ n ∈ Nat; m ∈ Nat ⟧ ⟹ Succ[m] -- Succ[n] = m -- n"*
lemma *diff_self_eq_0_nat [simp]: "m ∈ Nat ⟹ m -- m = 0"*

lemma *diff_diff_left_nat:* — Versión de la propiedad asociativa.
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat ⟧ ⟹ (a -- b) -- c = a -- (b + c)"
by *(rule diffInduct[of a b], auto)*

El lema anterior lo probamos por otro principio de inducción con una estructura diferente, que es apropiado para la diferencia de naturales. Si el principio *natInductE* es lineal, se dice que el principio *diffInduct* (definido en la teoría *Peano*) "se mueve por la diagonal" de los dos argumentos.

theorem *diffInduct:*
assumes *"n ∈ Nat"* **and** *"m ∈ Nat"*
and *"⋀ m. m ∈ Nat ⟹ P(m,0)"* **and** *"⋀ n. n ∈ Nat ⟹ P(0, Succ[n])"*
and *"⋀ m n. ⟦ m ∈ Nat; n ∈ Nat; P(m,n) ⟧ ⟹ P(Succ[m], Succ[n])"*
shows *"P(m,n)"*

lemma *Succ_diff_diff_nat [simp]:*
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat ⟧ ⟹ (Succ[a] -- b) -- Succ[c] = (a -- b) -- c"
by *(simp add: diff_diff_left_nat)*

lemma *diff_commute_nat:* — Versión de la propiedad conmutativa.
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ a -- b -- c = a -- c -- b"
by *(simp add: diff_diff_left_nat add_commute_nat)*

Las siguientes reglas cancelativas se prueban por inducción "lineal".

lemma *diff_add_inverse_nat [simp]: "⟦ b ∈ Nat; a ∈ Nat⟧ ⟹ (b + a) -- b = a"*
lemma *diff_cancel_nat [simp]: "⟦ c ∈ Nat; a ∈ Nat; b ∈ Nat⟧ ⟹ (c + a) -- (c + b) = a -- b"*
lemma *diff_add_0_nat [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ a -- (a + b) = 0"*

Las propiedades distributivas con respecto a la multiplicación se prueban con *diffInduct*. No se agregan al simplificador ya que a veces son usadas en el sentido contrario.

lemma *diff_mult_distrib_nat:* — Propiedad distributiva por derecha con respecto a $*$.
*"⟦ m ∈ Nat; n ∈ Nat; k ∈ Nat⟧ ⟹ (m -- n) * k = (m * k) -- (n * k)"*
lemma *diff_mult_distrib2_nat:* — Propiedad distributiva por izquierda con respecto a $*$.
*"⟦ m ∈ Nat; n ∈ Nat; k ∈ Nat⟧ ⟹ k * (m -- n) = (k * m) -- (k * n)"*

5.6. Propiedades adicionales

A continuación se muestran algunos de los resultados adicionales sobre la suma, resta y multiplicación que serán usados en las teorías que siguen. La lista de estos lemas es larga y sus pruebas no son complejas por lo que sólo se mostrarán los enunciados principales. Por cada uno de ellos, habrá lemas intermedios que fueron probados para llegar a esos resultados más generales. Y también habrá varias versiones de la misma propiedad: ya sea intercambiando los argumentos debido a la propiedad conmutativa, o ya sea adaptando la estructura para que sea agregada al simplificador o como regla de destrucción. Además, en general, un lema que tenga una condición no estricta ($\leq$) implica también que la condición puede ser estricta ($<$), o viceversa.

Monotonía de la suma

lemma *nat_add_left_cancel_leq [simp]:*
"⟦ k ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ (k + m ≤ k + n) = (m ≤ n)"

Grupos parcialmente ordernados

lemma *add_leq_mono:*
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat; d ∈ Nat; a ≤ b; c ≤ d⟧ ⟹ a + c ≤ b + d"

Las combinaciones obtenidas al cambiar $\leq$ por $<$ generan otras versiones de este lema. Además, de este lema se derivan resultados como:

lemma *trans_leq_add1: "⟦ a ≤ b; a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ a ≤ b + c"*
lemma *add_leqD1: "⟦ a + c ≤ b; a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ a ≤ b"*

y también los casos falsos:

lemma *leq_add_left_false [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ a + b ≤ a = FALSE"*

Más resultados sobre la resta

La resta es el inverso de la suma, en ecuaciones y en desigualdades:

lemma *le_add_diff_inverse [simp]:* "⟦ $n \leq m$; $m \in Nat$; $n \in Nat$ ⟧ ⟹ $n + (m -- n) = m$"
lemma *diff_leq_self [simp]:* "⟦ $m \in Nat$; $n \in Nat$ ⟧ ⟹ $m -- n \leq m$"

lemma *leq_iff_add:* — Conversión de ≤ a +.
"⟦ $m \in Nat$; $n \in Nat$ ⟧ ⟹ $m \leq n = (\exists\ k \in Nat: n = m + k)$"

Simplificación de la resta en desigualdades:

lemma *less_imp_diff_less [intro]:* "⟦ $a < b$; $a \in Nat$; $b \in Nat$; $k \in Nat$ ⟧ ⟹ $a -- k < b$"
lemma *diff_is_0_eq [simp]:* "⟦ $m \in Nat$; $n \in Nat$ ⟧ ⟹ $(m -- n = 0) = (m \leq n)$"
lemma *zero_less_diff [simp]:* "⟦ $m \in Nat$; $n \in Nat$ ⟧ ⟹ $(0 < n -- m) = (m < n)$"

Asociatividad entre la suma y la resta. Variando los signos de resta y suma, y los paréntesis, hay cuatro casos diferentes.

lemma *diff_add_assoc1 [simp]:*
"⟦ $c \leq b$; $a \in Nat$; $b \in Nat$; $c \in Nat$ ⟧ ⟹ $a + (b -- c) = (a + b) -- c$"

Transformación de resta en suma. Por el mismo motivo que para el lema de asociatividad, estas reglas se agregan al Simplificador ya que lo que se busca es agrupar los elementos positivos siempre que sea posible.

lemma *leq_diff_right_add_left_iff [simp]:*
"⟦ $k \leq b$; $a \in Nat$; $b \in Nat$; $k \in Nat$ ⟧ ⟹ $a \leq b -- k = (a + k \leq b)$"
lemma *diff_left_eq_add_right [simp]:*
"⟦ $k \leq a$; $a \in Nat$; $b \in Nat$; $k \in Nat$ ⟧ ⟹ $(a -- k = b) = (b + k = a)$"

Propiedades cancelativas de la resta:

lemma *diff_cancel_right_nat [simp]:* — Propiedad cancelativa por derecha.
"⟦ $c \leq a$; $c \leq b$; $a \in Nat$; $b \in Nat$; $c \in Nat$ ⟧ ⟹ $(a -- c = b -- c) = (a = b)$"
lemma *diff_cancel_left_nat [simp]:* — Propiedad cancelativa por izquierda.
"⟦ $a \leq c$; $b \leq c$; $a \in Nat$; $b \in Nat$; $c \in Nat$⟧ ⟹ $(c -- a = c -- b) = (a = b)$"

Monotonía de la multiplicación

lemma *mult_leq_mono:*
" ⟦ $a \leq b$; $c \leq d$; $a \in Nat$; $b \in Nat$; $c \in Nat$; $d \in Nat$ ⟧ ⟹ $a * c \leq b * d$"

De esta propiedad se derivan resultados como:

lemma *mult_leq_cancel_left [simp]:*
"⟦ $a \in Nat$; $b \in Nat$; $k \in Nat$⟧ ⟹ $(k * a \leq k * b) = (k = 0 \vee a \leq b)$"

5.7. Exponenciación

El módulo estándar *ProtoReals* de TLA$^+$ define a^b, donde $a > 0$, o $b > 0$, o $a \neq 0$ y $b \in Int$, a partir de los cuatro axiomas:

$$a^1 = a \tag{5.8}$$
$$a^{m+n} = a^m * a^n \quad \text{si } a \neq 0 \text{ y } m, n \in Int \tag{5.9}$$
$$0^b = 0 \quad \text{si } b > 0 \tag{5.10}$$
$$a^{b*c} = a^{(b^c)} \quad \text{si } a > 0 \tag{5.11}$$

más la condición de continuidad:

$$0 < a \text{ y } 0 < b \text{ implica } a^b \leq a^c. \tag{5.12}$$

Puede probarse que $a^0 = 1$ si $a \neq 0$ ($a = a^1 = a^0 * a^1 = a^0 * a$, por 5.8 y 5.9 y luego por las propiedades de la multiplicación). Pero 0^0 está indefinido con estos axiomas. En lugar de definir estas cinco propiedades como axiomas, definiremos la exponenciación recursivamente, y a partir de esta definición probaremos las propiedades.

$$a \hat{} 0 = 1 \tag{5.13}$$
$$a \hat{} \text{Succ}[n] = a * a \hat{} n \tag{5.14}$$

Para el caso de los naturales, consideraremos las condiciones $a > 0$ o $b > 0$. Ya que definiremos el operador con el esquema de recursión, cuyo caso base comienza en 0, usaremos la definición $0^0 = 1$ que se adapta a nuestro esquema. Se define entonces $a \hat{} n$ donde $a, n \in Nat$:

```
definition primrec_exp where
  "primrec_exp(a,n) ≡ LET f ≡ (CHOOSE g ∈ [Nat → Nat] :
                                  g[0] = 1 ∧ (∀x ∈ Nat : g[Succ[x]] = a * g[x]))
                      IN f[n]"

definition expn :: "[c,c] ⇒ c"        (infixl "^" 75)
where nat_exp_def: "⟦ a ∈ Nat; n ∈ Nat⟧ ⟹ (a ^ n) ≡ primrec_exp(a,n)"

lemma expIsNat [intro!,simp]:        — Propiedad de clausura.
  "⟦ a ∈ Nat; n ∈ Nat⟧ ⟹ a ^ n ∈ Nat"
lemma exp_a_0_nat [simp]:            — Caso base 5.13.
  "a ∈ Nat ⟹ a ^ 0 = 1"
lemma exp_a_Succ_nat [simp]:         — Paso recursivo 5.14.
  "⟦ a ∈ Nat; n ∈ Nat⟧ ⟹ a ^ Succ[n] = a * (a ^ n)"
```

Las propiedades 5.8 y 5.10 se prueban trivialmente con el Simplificador (a través del lema *exp_a_Succ_nat*) y por lo tanto no hace falta declarar los lemas correspondientes. A continuación probamos el resto de las propiedades y otras reglas más.

```
lemma exp_1_n_nat [simp]: "n ∈ Nat ⟹ 1 ^ n = 1"  by (induct set: Nat, simp_all)

lemma exp_add_nat:                   — Propiedad 5.9.
  assumes a: "a ∈ Nat"  and m: "m ∈ Nat"  and n: "n ∈ Nat"
```

```
  shows "a ^ (m + n) = (a ^ m) * (a ^ n)"
  using n m a apply (induct, simp_all)
  using m apply (induct, simp_all add: mult_assoc_nat)
proof (rule disjI1)
  fix n assume n: "n ∈ Nat" with a have "a * a * a ^ n = a * (a * a ^ n)"
    using mult_assoc_nat by simp
  with a n show "a * a * a ^ n = a * a ^ n * a"
    using mult_commute_nat by simp
qed

lemma exp_exp_nat [simp]:        — Propiedad 5.11.
  assumes a: "a ∈ Nat" and m: "m ∈ Nat" and n: "n ∈ Nat"
  shows "(a ^ m) ^ n = a ^ (m * n)"
  using n m a apply (induct, simp_all add: exp_add_nat)
  using m by (induct, auto simp: mult_commute_nat dest: sym)

lemma exp_leq_nat:        — Condición de continuidad, propiedad 5.12.
  assumes a: "a ∈ Nat" and b: "b ∈ Nat" and c: "c ∈ Nat" and pos: "0 < a" "0 < b"
  shows "b ≤ c ⟹ a ^ b ≤ a ^ c"
  using c a b pos apply (induct, simp_all)
proof -
  fix n assume "n ∈ Nat" "b ≤ n ⟹ a ^ b ≤ a ^ n" "b ≤ Succ[n]"
  from prems show "a ^ b ≤ a * a ^ n"
    apply (simp add: nat_leq_less[of b "Succ[n]" ], safe)
    using mult_leq_mono[of 1 a "a^ b" "a^ n" ] by simp_all
qed

lemma mult_distrib_exp_nat:        — Propiedad distributiva con respecto a la multiplicación.
  "⟦ n ∈ Nat; a ∈ Nat; b ∈ Nat⟧ ⟹ (a * b) ^ n = (a ^ n) * (b ^ n)"
proof (induct set: Nat, simp_all)
  fix n assume n: "n ∈ Nat" "a ∈ Nat" "b ∈ Nat" and 1: "(a * b) ^ n = a ^ n * b ^ n"
  from n have 2: "a * a ^ n * (b * b ^ n) = a * (b * (a ^ n) * (b ^ n))"
    using mult_ac_nat by simp
  with n have "a * b * (a ^ n * b ^ n) = ..."
    using mult_assoc_nat by simp
  with n 1 2 show "a * b * (a ^ n * b ^ n) = (a * a ^ n) * (b * b ^ n)" by simp
qed

lemma exp_is_0_iff [simp]:        — Evaluación de casos (a ^ n) = 0.
  assumes a: "a ∈ Nat" and n: "n ∈ Nat"
  shows "((a ^ n) = 0) = (a = 0 ∧ 0 < n)"
  using n a apply (induct, simp_all)
  using a by (induct, simp_all)

lemma exp_is_0_cases [dest]:        — Casos de (a ^ n) = 0, como regla de destrucción.
  "⟦ (a ^ n) = 0; a ∈ Nat; n ∈ Nat ⟧ ⟹ (a = 0 ∧ 0 < n)" by simp
```

Capítulo 6

Definición y aritmética de los números enteros

A partir de la formalización de los operadores aritméticos de los números naturales, expandimos las definiciones a los números enteros. Para esto, en primer lugar necesitamos definir la noción de número negativo. Al principio se consideró usar la definición de números enteros que se da en Isabelle/ZF a partir de clases de equivalencias [Pau06], pero fue descartada ya que define nuevos operadores aritméticos y lo que necesitamos aquí es que los operadores aritméticos para los naturales y enteros usen la misma sintaxis. Lo que haremos será expandir la signatura de los operadores sobre naturales para que incluyan los números negativos.

Para cada operador binario, ya sea las funciones aritméticas o las relaciones de orden, se consideran, para sus dos argumentos a y b, cuatro casos: las combinaciones que se forman al tomar cada uno como positivo o cero (es decir, que pertenece a Nat) o negativo. La combinación que se da cuando $a \in Nat$ y $b \in Nat$ es exactamente la definición del operador ya dada para los naturales. En los tres casos restantes, se definirá axiomáticamente el operador en función del caso conocido para los naturales. Al razonar con argumentos enteros, la idea es llevar la expresión con números negativos a la expresión equivalente con números naturales. Por lo tanto, las teorías sobre enteros tratarán principalmente sobre la simplificación a la teoría de los naturales, donde se desarrollan las reglas más avanzadas.

6.1. El operador menos unario

Definimos axiomáticamente el operador unario “menos”, que expresa la noción de número entero negativo. Su notación en TLA$^+$ es “$-$.”. De la misma manera que para los naturales debemos dar la sintaxis explícita de cada número negativo.

```
consts “minus” :: “c ⇒ c” (“-._” )

syntax
  “-.0” :: “c”  (“-.0” )
  “-.1” :: “c”  (“-.1” )
  ⋮
translations
  “-.0” ⇌ “-.(0)”
  “-.1” ⇌ “-.(1)”
  ⋮
```

Siempre que se define un sistema axiomático el objetivo es, en primer lugar, que el conjunto de axiomas sea correcto (*sound*) y completo; y luego, que el conjunto de axiomas sea mínimo. Para definir el operador *minus* damos los siguientes tres axiomas.

```
axioms
  neg0 [simp]: "-.0 = 0"
  neg_neg [simp]: "-.-.n = n"
  negNotInNat [simp]: "-.(Succ[n]) ∉ Nat"
```

Los dos primeros axiomas expresan la relación elemental entre los números negativos y los naturales. Necesitamos del axioma *negNotInNat* ya que, se puede saber si un elemento pertenece a *Nat*, pero no cuáles no pertenecen. Se puede probar fácilmente que los tres axiomas son correctos o consistentes entre sí, y también con el resto de los axiomas ya que el operador $-.$ aparece sólo aquí. A continuación, se muestran las reglas de simplificación del operador.

```
lemma eq_imp_negeq: "n = m ⟹ -.n = -.m"  by simp

lemma neg_neq_pos [simp]: "n ∈ Nat ⟹ -.(Succ[m]) = Succ[n] = FALSE"
proof (rule contradiction)
  assume "-.(Succ[m]) = Succ[n] ≠ FALSE"  and n: "n ∈ Nat"
  hence "-.(Succ[m]) = Succ[n]"  by auto
  hence "-.(Succ[m]) ∈ Nat"  using n by simp
  thus FALSE by simp
qed

lemma pos_neq_neg [simp]: "⟦ m ∈ Nat; n ∈ Nat ⟧ ⟹ Succ[m] = -.(Succ[n]) = FALSE"
  by (auto dest: eq_imp_negeq)

lemma minusInj [dest]: "-.n = -.m ⟹ n = m"  by (auto dest: eq_imp_negeq)
lemma minusInj_iff [simp]: "-.n = -.m = (n = m)"  by auto

lemma neg0_eq_0 [dest]: "-.n = 0 ⟹ (n = 0)"  by (auto dest: eq_imp_negeq)
lemma neg0_imp_0 [simp]: "-.n = 0 = (n = 0)"  by auto

lemma notneg0_imp_not0 [dest]: "-.n ≠ 0 ⟹ n ≠ 0"  by auto
lemma not0_imp_notNat [simp]: "⟦ n ∈ Nat; 0 < n⟧ ⟹ -.n ∉ Nat"
  using gt0_implies_Suc[of n] by auto

lemma negSuccNotZero [simp]: "n ∈ Nat ⟹ -.(Succ[n]) = 0 = FALSE"
  using minusInj by auto
lemma neg_in_nat_imp_false [dest]: "-.(Succ[n]) ∈ Nat ⟹ FALSE"  by simp

lemma neg_in_nat_is0 [simp]: "n ∈ Nat ⟹ -.n ∈ Nat = (n = 0)"  by (rule natCases, auto)
lemma neg_exists_nat: "-.n ∈ Nat ⟹ ∃ k ∈ Nat: n = -.k"  by force

lemma pos_eq_neg_false [simp]: "⟦ 0 < a; a ∈ Nat; b ∈ Nat⟧ ⟹ a = -.b = FALSE"
  by (rule natCases, auto)
lemma neg_eq_nat_is0 [dest]: "⟦ -.a = b; a ∈ Nat; b ∈ Nat⟧ ⟹ a = 0 ∧ b = 0"
  by (rule natCases, auto)
```

lemma *nat_eq_negself_is0 [simp]: "n ∈ Nat ⟹ n = -.n = (n = 0)"* **by** *auto*

6.2. El conjunto *Int*

Definimos el conjunto *Int* de los números enteros como la unión del conjunto de los naturales *Nat* y la forma negativa de todos los elementos de *Nat*. Por lo tanto, *Int* contiene los dos elementos equivalentes 0 y −.0.

definition *Int :: c* **where** *"Int ≡ Nat ∪ { -.n : n ∈ Nat }"*

lemma *natInInt [simp]: "n ∈ Nat ⟹ n ∈ Int"* **by** *(simp add: Int_def)*
lemma *intDisj: "n ∈ Int ⟹ n ∈ Nat ∨ n ∈ { -.n : n ∈ Nat}"* **by** *(auto simp: Int_def)*
lemma *negint_eq_int [simp]: "-.n ∈ Int = (n ∈ Int)"* **by** *(force simp: Int_def)*

La siguiente regla realiza un análisis de casos para un entero (es análoga a *natCases*):

lemma *intCases [case_names Positive Negative, cases set: Int]:*
 assumes *n: "n ∈ Int"* **and** *sc: "n ∈ Nat ⟹ P"* **and** *nsc: "⋀ m. ⟦ m ∈ Nat; n = -.m⟧ ⟹ P"*
 shows *"P"*
 using *prems* **unfolding** *Int_def* **by** *auto*

Análisis de casos sobre dos números enteros. También se define el lema *intCases3*, con tres argumentos enteros.

lemma *intCases2:*
 assumes *m: "m ∈ Int"* **and** *n: "n ∈ Int"*
  and *pp: "⋀ a b. ⟦ a ∈ Nat; b ∈ Nat; m = a; n = b⟧ ⟹ P(a,b)"*
  and *pn: "⋀ a b. ⟦ a ∈ Nat; b ∈ Nat; m = a; n = -.b⟧ ⟹ P(a,-.b)"*
  and *np: "⋀ a b. ⟦ a ∈ Nat; b ∈ Nat; m = -.a; n = b⟧ ⟹ P(-.a,b)"*
  and *nn: "⋀ a b. ⟦ a ∈ Nat; b ∈ Nat; m = -.a; n = -.b⟧ ⟹ P(-.a,-.b)"*
 shows *"P(m,n)"*
apply *(rule intCases[OF m])*
 apply *(rule intCases[OF n])* **apply** *(simp add: pp)* **apply** *(simp add: pn)*
 apply *(rule intCases[OF n])* **apply** *(simp add: np)* **apply** *(simp add: nn)*
done

lemma *trichotomy:* — Ley de tricotomía de los enteros.
 "n ∈ Int ⟹ n = 0 ∨ (∃ k ∈ Nat: n = Succ[k] ∨ n = -.(Succ[k]))"
 unfolding *Int_def* **apply** *(clarsimp, erule disjE)* **by** *(auto dest: not0_implies_Suc)*

6.3. Relaciones de orden

Aquí comenzamos a definir los operadores binarios con argumentos de números enteros. Empezaremos por la relación de orden estricta. El caso $a < b$ con $a \in Nat$ y $b \in Nat$ ya está definido en la teoría sobre naturales. Los otros tres casos son los siguientes:

axioms
 int_less_pn_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ a < -.b = FALSE"
 int_less_np_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ -.a < b = TRUE"

int_less_nn_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ -.a < -.b = (b < a)"

lemma *int_less_refl [simp]:*
"n ∈ Int ⟹ n < n = FALSE" **by** *(rule intCases, auto)*
lemma *neg_less_iff_less [simp]:*
"⟦ m ∈ Int; n ∈ Int ⟧ ⟹ -.n < -.m = (m < n)" **by** *(rule intCases2[of m n], simp_all)*
lemma *int_pos_is_nat [dest]:*
"⟦ 0 < n; n ∈ Int⟧ ⟹ n ∈ Nat ∧ 0 < n" **unfolding** *Int_def* **by** *auto*

La relación de orden no-estricta para enteros negativos se define de igual forma. Sólo hay que tener cuidado en el primer caso que si $a = b = 0$ el enunciado es falso.

axioms
int_leq_pn_def [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ a ≤ -.b = FALSE"
int_leq_np_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ -.a ≤ b = TRUE"
int_leq_nn_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ -.a ≤ -.b = (b ≤ a)"

lemma *int_leq_refl [intro,simp]: "n ∈ Int ⟹ n ≤ n"*
by *(rule intCases, auto)*

lemma *neg_le_iff_le [simp]: "⟦ m ∈ Int; n ∈ Int ⟧ ⟹ -.n ≤ -.m = (m ≤ n)"*
proof *(rule intCases2[of m n], simp_all)*
fix *x y* **assume** *"x ∈ Nat" "y ∈ Nat"*
thus *"y ≤ -.x = (x ≤ -.y)"* **by** *(auto elim!: natCases[of y] natCases[of x])*
qed

lemma *eq_leq_bothE:* — Reduce la igualdad sobre enteros a una doble desigualdad.
assumes *"m ∈ Int"* **and** *"n ∈ Int"* **and** *"m = n"* **and** *"⟦ m ≤ n; n ≤ m ⟧ ⟹ P"*
shows *"P"*
using *prems* **by** *simp*

6.4. Aritmética de los enteros

6.4.1. Suma

Para la suma, a diferencia de las relaciones de orden, el caso donde el primer argumento a es natural y el segundo argumento b es negativo no se simplifica a una expresión simple: es necesario analizar los casos de $a \leq b$. Por lo tanto no lo añadimos por defecto al Simplificador. El segundo caso donde a es negativo y b natural, se reduce al primer caso. El más sencillo es el tercero, que se reduce a una suma de naturales.

axioms
int_add_pn_def: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ a + -.b ≡ IF a ≤ b THEN -.(b -- a) ELSE a -- b"
int_add_np_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ -.a + b = b + -.a"
int_add_nn_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ -.a + -.b = -.(a + b)"

En cambio, sí podemos añadir al Simplificador estas variantes del axioma *int_add_pn_def*:

lemma *int_add_pn_case1 [simp]: "⟦ a ≤ b; a ∈ Nat; b ∈ Nat⟧ ⟹ a + -.b = -.(b -- a)"*
by *(simp add: int_add_pn_def)*

lemma *int_add_pn_case2 [simp]: "⟦ b < a; a ∈ Nat; b ∈ Nat⟧ ⟹ a + -.b = a -- b"*
by *(simp add: int_add_pn_def nat_less_antisym_leq_false)*
lemma *int_add_pn_case2' [simp]: "⟦ b ≤ a; a ∈ Nat; b ∈ Nat⟧ ⟹ a + -.b = a -- b"*
by *(force simp add: int_add_pn_def diff_is_0_eq')*

lemma *addIsInt [simp]:* — Propiedad de clausura.
"⟦ m ∈ Int; n ∈ Int⟧ ⟹ m + n ∈ Int" **by** *(rule intCases2[of m n], auto simp: int_add_pn_def)*

lemma *add_0_right_int [simp]:* — Elemento neutro.
"n ∈ Int ⟹ n + 0 = n" **by** *(rule intCases, auto)*

lemma *add_inverse_int [simp]:* — Elemento aditivo inverso.
"n ∈ Int ⟹ n + -.n = 0" **by** *(rule intCases2[of n "-.n"], auto)*

lemma *add_commute_int:* — Propiedad conmutativa.
"⟦ m ∈ Int; n ∈ Int⟧ ⟹ m + n = n + m"
by *(rule intCases2[of m n], auto simp: add_commute_nat)*

Para probar la propiedad cancelativa, probamos los siguientes 8 casos correspondientes a las tres variables, si son negativas o positivas. (En realidad, son menos casos por conmutatividad y, cuando son las tres positivas, ya está probado para los naturales.)

lemma *add_right_cancel_nnp_int [simp]:* — Caso *a* negativo, *b* negativo y *c* positivo.
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (c + -.a = c + -.b) = (a = b)"
unfolding *int_add_pn_def* **by** *(auto, simp add: nat_not_leq)*

lemma *add_right_cancel_ppn_int [simp]:*
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (a + -.c = b + -.c) = (a = b)"
unfolding *int_add_pn_def* **by** *(auto simp: trans_leq_add1, simp add: nat_not_leq)*

lemma *add_left_cancel_ppn_int [simp]:*
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ -.c + a = -.c + b ⟹ a = b"
by *(simp add: add_commute_nat)*

lemma *add_left_cancel_pnp_int [simp]:*
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (c + a = c + -.b) = (a = -.b)"
unfolding *int_add_pn_def*
using *add_assoc_nat[symmetric]* **by** *(auto simp add: nat_not_leq[unfolded less_def])*

lemma *add_left_cancel_pnn_int [simp]:*
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ a + -.c = -.(b + c) = (a = -.b)"
apply *(auto simp add: int_add_pn_def nat_not_leq)*
by *(simp add: add_commute_nat[of b c] add_assoc_nat[of c b a, symmetric])*

lemma *add_left_cancel_npp_int [simp]:*
"⟦ a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ c + -.a = b + c = (-.a = b)"
apply *(auto, drule sym)* **by** *(simp add: add_commute_nat[of b c])*

Ahora sí se puede probar fácilmente la propiedad conmutativa de los enteros, haciendo un análisis de casos de los tres argumentos:

```
lemma add_right_cancel_int [simp]:
  "⟦ a ∈ Int; b ∈ Int; c ∈ Int⟧ ⟹ a + c = b + c = (a = b)"
  by (rule intCases3[of a b c], auto simp add: add_commute_int[of a c] dest: sym)

lemma add_left_cancel_int [simp]:
  "⟦ a ∈ Int; b ∈ Int; c ∈ Int⟧ ⟹ c + a = c + b = (a = b)"  by (simp add: add_commute_int)
```

Para la propiedad asociativa hacemos lo mismo. Aquí se debe analizar por separado cada caso donde uno de los dos argumentos de la suma sea negativo, para reescribir el objetivo como una resta de naturales. Mostramos la prueba del primer caso.

```
lemma int_add_assoc1:
  "⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ m + (n + -.p) = (m + n) + -.p"
proof (rule nat_leq_less_cases[of p n], auto)
  assume "m ∈ Nat" "n ∈ Nat" "p ∈ Nat" "p ≤ n"
  thus "(m + n) -- p = m + n + -.p"
    by (simp add: trans_leq_add2)
next
  assume "m ∈ Nat" "n ∈ Nat" "p ∈ Nat" "n < p"
  hence 1: "n ≤ p"  by auto
  show "m + (n + -.p) = m + n + -.p"
    by (rule nat_leq_cases[of p "m + n" ], auto simp: prems 1 add_ac_nat)
qed

lemma int_add_assoc2: "⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ m + (n + -.p) = (m + -.p) + n"
lemma int_add_assoc3: "⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ m + -.(n + p) = m + -.n + -.p"
lemma int_add_assoc4: "⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ (n + p) + -.m = (n + -.m) + p"
lemma int_add_assoc5: "⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ -.m + (n + -.p) = n + -.m + -.p"
lemma int_add_assoc6: "⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ -.m + (p + -.n) = p + -.(m + n)"
```

Entonces, la prueba de la propiedad asociativa de los números enteros es:

```
lemma add_assoc_int:
  assumes m: "m ∈ Int"  and n: "n ∈ Int"  and p: "p ∈ Int"
  shows "m + (n + p) = (m + n) + p"
  apply(rule intCases3[OF m n p], simp_all)
    apply(rule add_assoc_nat, assumption+)
    apply(rule int_add_assoc1, assumption+)
    apply(rule int_add_assoc2, assumption+)
    apply(rule int_add_assoc3, assumption+)
    apply(rule int_add_assoc4, assumption+)
    apply(rule int_add_assoc5, assumption+)
    apply(rule int_add_assoc6, assumption+)
    apply(rule add_assoc_nat, assumption+)
done
```

Más simplificaciones de signos en la suma, sobre argumentos de naturales y enteros:

```
lemma minus_distrib_pn_nat [simp]: "⟦ m ∈ Nat; n ∈ Nat⟧ ⟹ -.(m + -.n) = n + -.m"
  by (rule nat_leq_cases[of n m], simp_all)
```

lemma *minus_distrib_pn_int [simp]: "⟦ m ∈ Int; n ∈ Int⟧ ⟹ -.(m + -.n) = n + -.m"*
by *(rule intCases2[of m n], simp_all add: add_commute_int)*
lemma *int_add_nn : "⟦ a ∈ Int; b ∈ Int⟧ ⟹ -.a + -.b = -.(a + b)"*
by *(rule intCases2[of a b], simp_all)*

6.4.2. Multiplicación

La multiplicación con números negativos resulta más sencilla que la suma ya que no aparecen expresiones donde sea necesario hacer análisis de casos.

axioms
int_mult_pn_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ a ∗ -.b = -.(a ∗ b)"
int_mult_np_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ -.a ∗ b = -.(a ∗ b)"
int_mult_nn_def [simp]: "⟦ a ∈ Nat; b ∈ Nat⟧ ⟹ -.a ∗ -.b = a ∗ b"

theorems *int_mult_def = int_mult_pn_def int_mult_np_def*

lemma *multIsInt [simp]:* — Propiedad de clausura.
"⟦ a ∈ Int; b ∈ Int⟧ ⟹ a ∗ b ∈ Int" **by** *(rule intCases2[of a b], simp_all)*

lemma *mult_0_right_int [simp]:* — Elemento neutro.
"a ∈ Int ⟹ a ∗ 0 = 0" **by** *(rule intCases[of a], simp_all)*
lemma *mult_0_left_int [simp]:*
"a ∈ Int ⟹ 0 ∗ a = 0" **by** *(rule intCases[of a], simp_all)*

lemma *mult_commute_int:* — Propiedad conmutativa.
"⟦ a ∈ Int; b ∈ Int⟧ ⟹ a ∗ b = b ∗ a"
by *(rule intCases2[of a b], simp_all add: mult_commute_nat)*

lemma *mult_1_right_int [simp]:* — Elemento de identidad.
"a ∈ Int ⟹ a ∗ 1 = a" **by** *(rule intCases[of a], simp_all)*
lemma *mult_1_left_int [simp]:*
"a ∈ Int ⟹ 1 ∗ a = a" **by** *(rule intCases[of a], simp_all)*

lemma *mult_assoc_int:* — Propiedad asociativa.
assumes *m: "m ∈ Int"* **and** *n: "n ∈ Int"* **and** *p: "p ∈ Int"*
shows *"m ∗ (n ∗ p) = (m ∗ n) ∗ p"*
by *(rule intCases3[OF m n p], simp_all add: mult_assoc_nat)*

Debido a que los axiomas reescriben la multiplicación de números negativos en expresiones simples, la propiedad asociativa se prueba fácilmente a partir de la asociatividad de los naturales. En cambio, para probar la propiedad distributiva, que involucra la suma de números negativos, debemos proceder como en la prueba de las propiedades cancelativa o asociativa de la suma. A continuación se muestra la propiedad distributiva por izquierda.

lemma *ppn_distrib_left_nat:* — Caso *m* positivo, *n* positivo, *p* negativo
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *p: "p ∈ Nat"*
shows *"m ∗ (n + -.p) = m ∗ n + -.(m ∗ p)"*
apply *(rule nat_leq_cases[OF p n])*

```
  apply (rule nat_leq_cases[of "m * p" "m * n" ])
  using prems apply (simp_all add: diff_mult_distrib2_nat)
done
```

Las pruebas de los dos lemas siguientes son similares a la de *ppn_distrib_left_nat*.

```
lemma npn_distrib_left_nat:
  "⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat ⟧ ⟹ -.m * (n + -.p) = -.(m * n) + m * p"
lemma nnp_distrib_left_nat:
  "⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat ⟧ ⟹ -.m * (-.n + p) = m * n + -.(m * p)"

lemma distrib_left_int:
  assumes m: "m ∈ Int"  and n: "n ∈ Int"  and p: "p ∈ Int"
  shows "m * (n + p) = (m * n + m * p)"
apply(rule intCases3[OF m n p],
  simp_all only: int_mult_def int_add_nn_def int_mult_nn_def addIsNat)
     apply(rule add_mult_distrib_left_nat, assumption+)
     apply(rule ppn_distrib_left_nat, assumption+)
     apply(simp add: add_commute_int, rule ppn_distrib_left_nat, assumption+)
     apply(simp only: int_add_nn_def multIsNat add_mult_distrib_left_nat)+
     apply(rule npn_distrib_left_nat, assumption+)
     apply(rule nnp_distrib_left_nat, assumption+)
     apply(simp only: add_mult_distrib_left_nat)
done
```

Con este resultado, probamos que el conjunto *Int* junto con los operadores + y $*$ forman un forman un *dominio entero*, es decir un anillo conmutativo[1] donde $1 \neq 0$.

6.4.3. Resta

La diferencia sobre enteros se define simplemente como la suma del complemento.

```
definition diff :: "[c,c] ⇒ c"        (infixl "-" 65)
where int_diff_def: "⟦ m ∈ Int; n ∈ Int⟧ ⟹ m - n = m + -.n"

lemma diffIsInt [simp]: — Propiedad de clausura
  "⟦ m ∈ Int; n ∈ Int⟧ ⟹ m - n ∈ Int"  by (simp add: int_diff_def)
lemma diff_neg_is_add [simp]:
  "⟦ m ∈ Int; n ∈ Nat⟧ ⟹ m - -.n = m + n"  by (simp add: int_diff_def)
lemma diff_0_right_int [simp]:
  "m ∈ Int ⟹ m - 0 = m"  by (simp add: int_diff_def)
lemma diff_0_left_int [simp]:
  "n ∈ Int ⟹ 0 - n = -.n"  by (simp add: int_diff_def)
lemma diff_self_eq_0_int [simp]:
  "m ∈ Int ⟹ m - m = 0"  by (simp add: int_diff_def)
lemma neg_diff_is_diff [simp]:
```

[1] Un *anillo conmutativo* es la estructura algebraica $(A, +, *)$ tal que $(A, +)$ forma un grupo abeliano, $(A, *)$ forman un monoide conmutativo y se cumple la propiedad distributica de + y $*$. Un *grupo abeliano* $(A, +)$ satisface las propiedades de clausura, existencia de elemento identidad, conmutatividad, asociatividad y existencia de aditivo inverso. Un *monoide conmutativo* $(A, *)$ satisface las propiedades de clausura, asociatividad, existencia de identidad y conmutatividad.

```
  "⟦ m ∈ Int; n ∈ Int⟧ ⟹ -.(m - n) = n - m"  by (simp add: int_diff_def)
lemma neg_diff_is_neg_sum [simp]:
  "⟦ m ∈ Int; n ∈ Int⟧ ⟹ -.m - n = -.(m + n)"  by (simp add: int_diff_def int_add_nn)

lemma add_diff_assoc1_int:      — Propiedad asociativa de la resta y la suma, caso 1.
  "⟦ a ∈ Int; b ∈ Int; c ∈ Int⟧ ⟹ a + (b - c) = a + b - c"
  by (simp add: int_diff_def add_assoc_int)
lemma add_diff_assoc2_int:      — Propiedad asociativa de la resta y la suma, caso 2.
  "⟦ a ∈ Int; b ∈ Int; c ∈ Int⟧ ⟹ -.a + (b + -.c) = b + -.a + -.c"
  by (simp add: add_assoc_int, simp add: add_commute_int)
```

6.4.4. Exponenciación

Antes de hacer de definir la exponenciación a^n con base a negativa, se necesitan reglas para razonar sobre el predicado isEven ("es par").

El predicado "es par"

(Esta sección, en realidad, forma parte de la teoría sobre números naturales.)

```
definition isEven where "n ∈ Nat ⟹ isEven(n) ≡ ∃ x ∈ Nat : n = x + x"

lemma isEven_0 [simp]: "isEven(0)"
lemma evenSS_eq_even [simp]: "n ∈ Nat ⟹ isEven(Succ[Succ[n]]) = isEven(n)"
lemma nat_is_even_or_not: "n ∈ Nat ⟹ isEven(n) ∨ ¬ isEven(n)"
```

Uno de los principales resultados obtenidos es el siguiente, que establece que un número par no es igual a un número impar.

```
lemma odd_is_even_false [simp]:
  assumes x: "x ∈ Nat"  and y: "y ∈ Nat"  shows "(Succ[y + y] = x + x) = FALSE"
  apply (rule nat_leq_cases[of x y])
  using prems apply (simp_all add: nat_not_leq nat_leq_less)
proof (auto)
  assume h1: "x < y"  and h2: "Succ[y + y] = x + x"
  from x y x y h1 h1 have "x + x < y + y"  by (rule add_less_mono)
  with y h2[symmetric] show "FALSE"  by simp
next
  assume h1: "y < x"  and h2: "Succ[y + y] = x + x"
  from x y h1 have "Succ[y] + y < x + x"
    using nat_Succ_leq_iff_less by (simp add: add_leq_less_mono del: add_Succ_left_nat)
  with y h2[symmetric] show "FALSE"  by simp
qed
```

Esta regla permite probar los siguientes lemas:

```
lemma even_is_odd_false [simp]: "⟦ x ∈ Nat; y ∈ Nat⟧ ⟹ (x + x = Succ[y + y]) = FALSE"
lemma isEven_add_Sxx_false [simp]: "n ∈ Nat ⟹ isEven(Succ[n + n]) = FALSE"
lemma isEven_imp_notSucc [simp]: "⟦ isEven(n); n ∈ Nat⟧ ⟹ isEven(Succ[n]) = FALSE"
lemma evenS_imp_even_false [dest]: "⟦ isEven(Succ[n]); n ∈ Nat⟧ ⟹ isEven(n) = FALSE"
```

lemma *evenS_even_false [dest]: "⟦ isEven(Succ[n]); isEven(n); n ∈ Nat⟧ ⟹ FALSE"*
lemma *notEven_is_evenS [dest!]: "⟦ ¬ isEven(n); n ∈ Nat ⟧ ⟹ isEven(Succ[n])"*
lemma *notEvenS_is_even [dest!]: "⟦ ¬ isEven(Succ[n]); n ∈ Nat⟧ ⟹ isEven(n)"*

lemma *isEven_cases [case_names even odd]:*
assumes *n: "n ∈ Nat"* **and** *even: "isEven(n) ⟹ P"* **and** *odd: "isEven(Succ[n]) ⟹ P"*
shows *"P"*
apply *(cases "isEven(n)")* **using** *prems* **by** *auto*

Razonamiento sobre la suma:

lemma *even_plus_even_is_even [simp]:*
"⟦ isEven(m); isEven(n); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven(m + n)"
lemma *evenS_plus_even_is_evenS [simp]:*
"⟦ isEven(Succ[m]); isEven(n); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven(Succ[m + n])"
using *even_plus_even_is_even[of "Succ[m]" n]* **by** *simp*
lemma *evenS_plus_even_is_even_false1 [simp]:*
"⟦ isEven(Succ[m]); isEven(n); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven(m + n) = FALSE"
by *(auto dest: evenS_plus_even_is_evenS)*

Razonamiento sobre la multiplicación:

lemma *even_imp_mult_even [simp]:*
"⟦ isEven(m); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven (m ∗ n)"
apply *(simp add: isEven_def, clarsimp)* **by** *(auto simp: add_mult_distrib_right_nat)*
lemma *multEven_imp_either_even [dest]:*
"⟦ isEven(m ∗ n); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven(m) ∨ isEven(n)"
apply *(rule natCases[of m], simp_all)* **by** *(cases "isEven(n)" , auto)*

Exponenciación con base negativa

Ahora, con el predicado "es par" ya formalizado, se puede definir la exponenciación cuya base es un número entero negativo. Lo hacemos a través del siguiente axioma.

axioms *exp_neg: "⟦ a ∈ Nat; n ∈ Nat⟧ ⟹ -.a ^ n = (IF isEven(n) THEN a ^ n ELSE -.(a ^ n))"*

Con este axioma más la exponenciación para naturales definimos a^n donde $a \in Int$ y $n \in nat$. El caso donde $n \in Int$ deberá definirse para los números reales.

lemma *expIsInt [intro!,simp]:* — Propiedad de clausura.
"⟦ a ∈ Int; n ∈ Nat⟧ ⟹ a ^ n ∈ Int" **by** *(auto elim!: intCases simp: exp_neg nat_is_even_or_not)*
lemma *exp_neg1_even [simp]:*
"n ∈ Nat ⟹ -.1 ^ (n ∗ 2) = 1" **by** *(simp add: exp_neg)*
lemma *exp_neg_even [simp]:*
"⟦ a ∈ Nat; n ∈ Nat⟧ ⟹ -.a ^ (n ∗ 2) = a ^ (n ∗ 2)" **by** *(simp add: exp_neg)*
lemma *exp_neg_1 [simp]:*
"a ∈ Nat ⟹ -.a ^ 1 = -.a" **by** *(simp add: exp_neg)*

Los siguientes lemas se prueban por análisis de casos del predicado *isEven* luego de expandir la definición de *exp*. Las pruebas de los casos se terminan por el razonamiento de *isEven* ya agregado a los métodos automáticos.

```
lemma exp_neg_add:      — Corresponde al axioma 5.9 de la sección˜5.7.
  "⟦ a ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ -.a ^ (m + n) = (-.a ^ m) * (-.a ^ n)"
  apply (simp add: exp_neg)
  apply (rule isEven_cases[of "(m + n)" ], simp+)
   apply (rule isEven_cases[of "m" ], simp+)
    apply (rule isEven_cases[of "n" ], simp_all add: exp_add_nat)
   apply (rule isEven_cases[of "n" ], simp+, force)
  apply (rule isEven_cases[of "n" ], simp+)
   apply (rule isEven_cases[of "m" ], simp+, force)
   apply (rule isEven_cases[of "m" ], simp+, force)
   using even_plus_even_is_even[of "Succ[m]" "Succ[n]" ] apply force
done

lemma exp_exp_neg [simp]:      — Corresponde al axioma 5.11 de la sección˜5.7.
  "⟦ a ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ (-.a ^ m) ^ n = -.a ^ (m * n)"
  apply (simp add: exp_neg)
  apply (cases "isEven(m)" , simp_all add: condElse)
   apply (cases "isEven(m * n)" , simp_all add: condElse)
    apply (safe dest!: multEven_imp_either_even)
    apply (force, simp_all add: exp_neg)
   apply (cases "isEven(n)" , simp_all add: condElse)
   using even_imp_mult_even[of n "Succ[m]" ]
    even_imp_mult_even[of "Succ[m]"  n] apply simp
done
```

6.5. División y módulo sobre los números naturales y enteros

El Algoritmo de División es usado para calcular la división entera. Básicamente describe el proceso de dividir un número entero (a) por otro entero positivo (b) para obtener el cociente (q) y el resto (r). Este es el conocido proceso por el cual los niños aprenden a dividir en la escuela primaria. El Algoritmo de División no es propiamente un algoritmo, es un teorema y se declara a continuación.

Teorema [Algoritmo de División]. Dados dos enteros a y b, donde $b > 0$. Existen enteros únicos q y r, llamados *cociente* y *resto* respectivamente, tales que

$$a = q*b + r \qquad \text{y} \qquad 0 \leq r \wedge r < b$$

Tomando $q = a \div b$ y $r = a \ \% \ b$, esta declaración del teorema es exactamente la definición de los operadores de TLA$^+$ de división ($\div$) y módulo (%) dada en la sección 2.2.3 sobre los módulos estándar sobre números de TLA$^+$. Hay que tener en cuenta que, con esta definición, $-.a \div b$ donde $a, b \in Nat$ no es igual a $-.(a \div b)$ como uno podría suponer, ya que no satisfacen las ecuaciones del teorema. Por ejemplo, $-.5 \div 3 = -.2$ y $-.5 \ \% \ 3 = 1$ sí satisfacen las ecuaciones

$$-.5 = (-.5 \div 3)*3 + (-.5 \ \% \ 3) \quad \text{y} \quad 0 \leq (-.5 \ \% \ 3) \wedge (-.5 \ \% \ 3) < 3.$$

Prueba del Algoritmo de División

La prueba del Algoritmo de División nos llevará a la definición de los operadores buscados. La prueba consiste en dos partes: primero se prueba la existencia de q y de r dados a y b, y luego se prueba

que son únicos. Trataremos principalmente con dos casos: $a, q \in Nat$ (caso 1) y $a, q \in \{-.n : n \in Nat\}$ (caso 2). Si a y q tienen distintos signos entonces significa que ambos son 0 ($0 = -.0$) (casos 3 y 4). El caso 1 corresponde, obviamente, a la división entre naturales. Muchas de las pruebas de esta sección son largas, por lo que sólo mostrarán las más importantes y el resto se explicarán informalmente en caso de que sea necesario. Las pruebas completas pueden encontrarse en el apéndice A.

Comenzamos definiendo la relación característica *divmod_rel* entre las cuatro variables involucradas.

```
definition divmod_rel where
  "divmod_rel(a,b,q,r) ≡ a = q ∗ b + r ∧ (0 < b) ∧ (0 ≤ r) ∧ (r < b)"
```

Los siguientes lemas prueban que dados los enteros a y b (positivo), existen q y r que satisfacen la relación *divmod_rel* para los dos casos tratados. Por lo tanto, expresan que la relación *divmod_rel* es total. Se mostrarán solamente las pruebas del caso 1; la del caso 2 es similar.

```
lemma divmod_rel_ex_case1:
  assumes "a ∈ Nat" "b ∈ Nat" "0 < b"
  obtains q r where "q ∈ Nat" "r ∈ Nat" "divmod_rel(a,b,q,r)"
proof -
  have "∃ q,r ∈ Nat : a = q * b + r ∧ r < b"
  using a proof (induct)
    case 0
    from b pos have "0 = 0 * b + 0 ∧ 0 < b" by simp
    then show ?case by blast
  next
    fix a'
    assume a': "a' ∈ Nat" and ih: "∃ q, r ∈ Nat : a' = q*b + r ∧ r < b"
    from ih obtain q' r'
      where h1: "a' = q' * b + r'" and h2: "r' < b"
      and q': "q' ∈ Nat" and r': "r' ∈ Nat" by blast
    show "∃ q, r ∈ Nat : Succ[a'] = q * b + r ∧ r < b"
    proof (cases "Succ[r'] < b" )
      case True
      from h1 h2 a' q' b r' have "Succ[a'] = q' * b + Succ[r']" by simp
      with True q' r' show ?thesis by blast
    next
      case False
      with b r' have "b ≤ Succ[r']" by (simp add: nat_not_less)
      with r' b h2 have "b = Succ[r']" by (intro nat_leq_antisym, simp+)
      with h1 a' q' r' have "Succ[a'] = Succ[q'] * b + 0" by (simp add: add_ac_nat)
      with pos q' show ?thesis by blast
    qed
  qed
  with pos that show ?thesis by (auto simp add: divmod_rel_def)
qed

lemma divmod_rel_ex_case2:
  assumes "a ∈ Nat" "b ∈ Nat" "0 < b"
  obtains q r where "q ∈ Nat" "r ∈ Nat" "divmod_rel(-.a,b,-.q,r)"
```

La notación **obtains-where** de Isabelle expresa que el objetivo a probar es (en el primer caso):

(⋀q r. ⟦ q ∈ Nat; r ∈ Nat; divmod_rel(?a, ?b, q, r) ⟧ ⟹ ?thesis) ⟹ ?thesis

Estos lemas se prueban por inducción. La idea general de las pruebas del paso inductivo es analizar los casos en que el resto es igual a 0 y distinto de 0. En el caso 1, se prueba tratando por separado el caso $r \leq b$, es decir cuando el resto r es menor que el dividendo b y cuando el resto es $r = b$ (y por lo tanto $r = 0$ y q se incrementa en 1 para satisfacer la relación $a = \text{Succ}[q] * b + 0$). En el caso 2, se tratan los casos de $r > 0$, es decir se prueba que el paso inductivo se cumple sin variar q y con $r - 1$ cuando $r > 0$, y que se cumple con $-.(\text{Succ}[q])$ y sin variar r cuando $r = 0$.

La prueba del teorema del Algoritmo de División se completa con la prueba de unicidad de las soluciones de *divmod_rel*. A continuación mostramos los lemas de unicidad de q (los lemas sobre la unicidad de r son análogos en ambos casos).

```
lemma divmod_rel_unique_div_case1:
  assumes "divmod_rel(a,b,q,r)" "divmod_rel(a,b,q',r')"
   "a ∈ Nat" "b ∈ Nat" "q ∈ Nat" "r ∈ Nat" "q' ∈ Nat" "r' ∈ Nat"
  shows "q = q'"
proof -
  from b 1 have pos: "0 < b"  and aqr: "a = q*b+r"  and rb: "r < b"
    by (auto simp add: divmod_rel_def)
  from b 2 have aqr': "a = q'*b+r'"  and rb': "r' < b"
    by (auto simp add: divmod_rel_def)
  {
    fix x y x' y'
    assume nat: "x ∈ Nat"  "y ∈ Nat"  "x' ∈ Nat"  "y' ∈ Nat"
      and eq: "x*b + y = x'*b + y'"  and less: "y' < b"
    have "x ≤ x'"
    proof (rule contradiction)
      assume "¬ (x ≤ x')"
      with nat have "x' < x"  by (simp add: nat_not_leq)
      with nat obtain k where k: "k ∈ Nat"  "x = Succ[x'+k]"
        by (auto simp add: less_iff_Succ_add)
      with eq nat b have "x'*b + (k*b + b + y) = x'*b + y'"
        by (simp add: add_mult_distrib_right_nat add_assoc_nat)
      with nat k b have "k*b + b + y = y'"  by simp
      with less k b nat have "(k*b + y) + b < b"  by (simp add: add_ac_nat)
      with k b nat show "FALSE"  by simp
    qed
  }
  from this[OF q r q' r'] this[OF q' r' q r] q q' aqr aqr' rb rb'
  show ?thesis by (intro nat_leq_antisym, simp+)
qed

lemma divmod_rel_unique_mod_case2:
  assumes "divmod_rel(-.a,b,-.q,r)" "divmod_rel(-.a,b,-.q',r')"
   "a ∈ Nat" "b ∈ Nat" "q ∈ Nat" "r ∈ Nat" "q' ∈ Nat" "r' ∈ Nat"
  shows "q = q'"
```

Definición de los operadores de división y módulo

Instanciamos las soluciones de la relación característica *divmod_rel* usando el operador CHOOSE. Definimos el predicado *divmod* que devuelve una tupla con los resultados. Usando los lemas de la prueba del Algoritmo de División, en *divmodPairEx* probamos que los resultados existen, y en *divmod_unique* probamos que son únicos.

```
definition divmod where
  "divmod(a,b) ≡ CHOOSE z ∈ Int × Nat : divmod_rel(a,b,z[1],z[2])"

lemma divmodPairEx:        — Existencia de una tupla resultante de divmod
  assumes a: "a ∈ Int" and b: "b ∈ Nat" and pos: "0 < b"
  shows "∃ z ∈ Int × Nat : divmod_rel(a,b,z[1],z[2])"
proof (rule intCases[OF a])
  assume a: "a ∈ Nat"
  from a b pos obtain q r where "r ∈ Nat" "q ∈ Nat" "divmod_rel(a,b,q,r)"
    by (rule divmod_rel_ex_case1)
  thus ?thesis by (auto intro!: inProdI)
next
  fix a'
  assume a': "a' ∈ Nat" and 1: "a = -.a'"
  obtain q r where "q ∈ Nat" "r ∈ Nat" "divmod_rel(-.a',b,-.q,r)"
    by (rule divmod_rel_ex_case2[OF a' b pos])
  thus ?thesis
    using prems 1 apply auto by (rule_tac x="⟨-.q,r⟩" in bExI, auto)
qed


lemma divmodInIntNat [intro!,simp]:        — Clausura del caso general
  "[[ a ∈ Int; b ∈ Nat; 0 < b ]] ⟹ divmod(a,b) ∈ Int × Nat"
lemma divmodInNatNat:
  "[[ a ∈ Int; b ∈ Nat; 0 < b ]] ⟹ divmod(a,b) ∈ Nat × Nat"
lemma divmodInNegNat:
  "[[ a ∈ Int; b ∈ Nat; 0 < b ]] ⟹ divmod(-.a,b) ∈ {-.n : n ∈ Nat} × Nat"

lemma divmod_unique_case1:
  "[[ divmod_rel(a,b,q,r); a ∈ Nat; b ∈ Nat; 0 < b; q ∈ Nat; r ∈ Nat ]]
  ⟹ divmod(a,b) = ⟨q,r⟩"
lemma divmod_unique_case2:
  "[[ divmod_rel(-.a,b,-.q,r); a ∈ Nat; b ∈ Nat; 0 < b; q ∈ Nat; r ∈ Nat ]]
  ⟹ divmod(-.a,b) = ⟨-.q,r⟩"
lemma divmod_unique_case3:
  "[[ divmod_rel(a,b,-.q,r); a ∈ Nat; b ∈ Nat; 0 < b; q ∈ Nat; r ∈ Nat ]]
  ⟹ divmod(a,b) = ⟨-.q,r⟩"
lemma divmod_unique_case4:
  "[[ divmod_rel(-.a,b,q,r); a ∈ Nat; b ∈ Nat; 0 < b; q ∈ Nat; r ∈ Nat ]]
  ⟹ divmod(-.a,b) = ⟨q,r⟩"

lemma divmod_unique:        — Caso general de unicidad de divmod.
```

"⟦ divmod_rel(a,b,q,r); a ∈ Int; b ∈ Nat; 0 < b; q ∈ Int; r ∈ Nat ⟧ ⟹ divmod(a,b) = ⟨q,r⟩"
apply *(rule intCases2[of a q])*
by *(simp_all add: divmod_unique_case1 divmod_unique_case2*
divmod_unique_case3 divmod_unique_case4)

Finalmente, podemos definir los operadores ÷ y % como los elementos 1 y 2, respectivamente, de la tupla resultante de *divmod*.

defs
div_def: "⟦ a ∈ Int; b ∈ Nat; 0 < b ⟧ ⟹ a ÷ b ≡ divmod(a,b)[1]"
mod_def: "⟦ a ∈ Int; b ∈ Nat; 0 < b ⟧ ⟹ a % b ≡ divmod(a,b)[2]"

A continuación se prueban para ÷ la propiedad de clausura general (*divIsInt*), de clausura para el caso de que el dividendo sea natural (*divIsNat*) o entero negativo (*divIsNeg*), y la propiedad de clausura de% (*modIsNat*).

lemma *divIsInt [intro!, simp]: "⟦ a ∈ Int; b ∈ Nat; 0 < b ⟧ ⟹ a ÷ b ∈ Int"*
lemma *divIsNat [intro!, simp]t: "⟦ a ∈ Nat; b ∈ Nat; 0 < b ⟧ ⟹ a ÷ b ∈ Nat"*
lemma *divIsNeg [intro!, simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b ⟧ ⟹ -.a ÷ b ∈ {-.n : n ∈ Nat}"*
lemma *modIsNat [intro!, simp]: "⟦ a ∈ Int; b ∈ Nat; 0 < b ⟧ ⟹ a ÷ b ∈ Nat"*

lemma *divmod_div_mod:* — ÷ y% son el resultado de *divmod*
"⟦ a ∈ Int; b ∈ Nat; 0 < b ⟧ ⟹ divmod(a,b) = ⟨a ÷ b, a % b⟩"
lemma *divmod_div_mod_nat:* — mismo resultado para naturales
"⟦ a ∈ Nat; b ∈ Nat; 0 < b ⟧ ⟹ divmod(a,b) = ⟨a ÷ b, a % b⟩"

lemma *divmod_rel_div_mod:* — ÷ y% satisfacen *divmod_rel*
"⟦ a ∈ Int; b ∈ Nat; 0 < b ⟧ ⟹ divmod_rel(a, b, a ÷ b, a % b)"

lemma *div_unique:* — el resultado de la división es único
"⟦ divmod_rel(a,b,q,r); a ∈ Int; b ∈ Nat; 0 < b; q ∈ Int; r ∈ Nat ⟧ ⟹ a ÷ b = q"
lemma *mod_unique:* — el resultado del módulo es único
"⟦ divmod_rel(a,b,q,r); a ∈ Int; b ∈ Nat; 0 < b; q ∈ Int; r ∈ Nat ⟧ ⟹ a ÷ b = r"
lemma *mod_less_divisor:* — el módulo es siempre menor que el divisor
"⟦ a ∈ Int; b ∈ Nat; 0 < b ⟧ ⟹ a % b < b"

Directamente del lema *divmod_rel_div_mod* podemos probar el lema *mod div int equality1*:

lemma *mod_div_int_equality1 [simp]: "⟦ a ∈ Int; b ∈ Nat; 0 < b⟧ ⟹ (a ÷ b) ∗ b + a % b = a"*
lemma *mod_div_int_equality2 [simp]: "⟦a ∈ Int; b ∈ Nat; 0 < b⟧ ⟹ b ∗ (a ÷ b) + a % b = a"*
lemma *mod_div_int_equality3 [simp]: "⟦ a ∈ Int; b ∈ Nat; 0 < b⟧ ⟹ a % b + (a ÷ b) ∗ b = a"*
lemma *mod_div_int_equality4 [simp]: "⟦ a ∈ Int; b ∈ Nat; 0 < b⟧⟹ a % b + b ∗ (a ÷ b) = a"*

Podemos dar las fórmulas de lo que sería el cálculo recursivo sobre el primer argumento de *divmod* para los naturales:

lemma *divmod_base_case1:*
"⟦ a ∈ Nat; b ∈ Nat; a < b ⟧ ⟹ divmod(a,b) = ⟨0, a⟩"
lemma *divmod_step_case1:*
"⟦ a ∈ Nat; b ∈ Nat; b ≤ a ⟧ ⟹ divmod(a,b) = ⟨Succ[(a -- b) ÷ b], (a -- b) % b⟩"

Por lo tanto, las ecuaciones "recursivas" de ÷ y % son:

lemma *div_nat_less [simp]: "⟦ a ∈ Nat; b ∈ Nat; a < b ⟧ ⟹ (a ÷ b) = 0"*
lemma *div_nat_geq: "⟦ a ∈ Nat; b ∈ Nat; b ≤ a ⟧ ⟹ (a ÷ b) = Succ[(a -- b) ÷ b]"*

lemma *mod_nat_less [simp]: "⟦ a ∈ Nat; b ∈ Nat; a < b ⟧ ⟹ (a % b) = a"*
lemma *mod_nat_geq: "⟦ a ∈ Nat; b ∈ Nat; b ≤ a ⟧ ⟹ (a % b) = (a -- b) % b"*
declare *mod_nat_geq[symmetric, simp]*

Propiedades de ÷ y % sobre naturales

Para este tipo de propiedades donde hay sumas y multiplicaciones existen varias versiones del mismo lema, con las que es conveniente contar en el simplificador. La primera versión es probada normalmente y agregada al simplificador. Las siguientes son probadas trivialmente a partir de la primera y la propiedad conmutativa correspondiente. De ahora en más sólo mostraremos la primera versión del lema.

Por ahora, podemos probar que la división y el módulo distribuye en la suma si uno de los sumandos es múltiplo del divisor *n*:

lemma *div_nat_mult_self1 [simp]:*
"⟦ q ∈ Nat; m ∈ Nat; n ∈ Nat; 0 < n ⟧⟹ (q + m ∗ n) ÷ n = m + q ÷ n"
lemma *mod_nat_mult_self1 [simp]:*
"⟦ q ∈ Nat; m ∈ Nat; n ∈ Nat; 0 < n ⟧⟹ (q + m ∗ n) % n = q % n"

A partir de estos dos lemas generales, e instanciando sus variables en 0 o 1, podemos probar fácilmente los siguientes lemas.

lemma *div_nat_mult_self1_is_id [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a ∗ b) ÷ b = a"*
lemma *mod_nat_mult_self1_is_0 [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a ∗ b) % b = 0"*
lemma *div_nat_by_1 [simp]: "a ∈ Nat ⟹ a ÷ 1 = a"*
lemma *mod_nat_by_1 [simp]: "a ∈ Nat ⟹ a % 1 = 0"*
lemma *div_nat_self [simp]: "⟦ b ∈ Nat; 0 < b⟧ ⟹ b ÷ b = 1"*
lemma *mod_nat_self [simp]: "⟦ b ∈ Nat; 0 < b⟧ ⟹ b % b = 0"*
lemma *div_add_self1_case1 [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a + b) ÷ b = a ÷ b + 1"*
lemma *div_Succ_add_self1_case1 [simp]:*
"⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ Succ[a + b] ÷ b = Succ[a] ÷ b + 1"
lemma *mod_nat_add_self1 [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a + b) % b = a % b"*
lemma *mod_nat_Succ_add_self1 [simp]:*
"⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ Succ[a + b] % b = Succ[a] % b"

La división y el módulo se distribuyen sobre la resta de naturales, en los siguientes casos:

lemma *div_diff_self_less_nat [simp]:*
"⟦ r ∈ Nat; n ∈ Nat; 0 < n; 0 < r; r < n ⟧ ⟹ (n -- r) ÷ n = 0"
lemma *mod_diff_self_less_nat [simp]:*
"⟦ r ∈ Nat; n ∈ Nat; 0 < n; 0 < r; r < n ⟧ ⟹ (n -- r) % n = n -- r"
lemma *div_diff_self_Succ_less_nat [simp]:*
"⟦ r ∈ Nat; n ∈ Nat; 0 < n; Succ[r] < n ⟧ ⟹ (n -- Succ[r]) ÷ n = 0"
lemma *mod_diff_self_Succ_less_nat [simp]:*
"⟦ r ∈ Nat; n ∈ Nat; 0 < n; Succ[r] < n ⟧ ⟹ (n -- Succ[r]) % n = n -- Succ[r]"

Evaluación de casos $a \div b = k$ y $a \ \% \ b = k$, donde $a, b, k \in Nat$:

```
lemma mod0_imp_ex_divmod_nat:
  "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ a % b = 0 ⇒ (∃ q ∈ Nat : divmod_rel(a,b,q,0))"
lemma mod0_imp_dvd_nat [dest]:
  "⟦ a % b = 0; a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ ∃ q ∈ Nat : a = q ∗ b"
lemma mod_not0_imp_ex_divmod_nat:
  "⟦ a % b ≠ 0; a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ ∃ q, r ∈ Nat : divmod_rel(a,b,q,r) ∧ 0 < r"
lemma mod0_div_mult_self [simp]:
  "⟦ a % b = 0; a ∈ Nat; b ∈ Nat; 0 < b ⟧ ⟹ (a ÷ b) ∗ b = a"
lemma div_mult_self_leq [simp]:
  "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a ÷ b) ∗ b ≤ a"
lemma div_mult_self_gt [intro]:
  "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ a < a ÷ b ∗ b + b"
lemma div_to_leq_less_case1:
  "⟦ a ÷ b = k; a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ k ∗ b ≤ a ∧ a < Succ[k] ∗ b"
```

Relación entre la división entera sobre naturales y sobre enteros negativos

Hasta ahora hemos probado propiedades de $a \div b$ y $a \ \% \ b$ donde $a, b \in Nat$. Para los casos $a' \div b$ y $a' \ \% \ b$ donde $a' \in \{-.n : n \in Nat\}$ buscaremos la relación que existe con los resultados para los naturales. Luego, cuando sea necesario, podremos llevar el análisis de la división de enteros negativos a naturales. La relación que encontramos es:

$$\begin{array}{lcl} -a \div b & = & -((a-1) \div b) - 1 \\ -a \ \% \ b & = & \text{IF } (a-1) \ \% \ b = 0 \text{ THEN } 0 \text{ ELSE } (b - (a-1) \ \% \ b) \end{array}$$

donde $a, b \in Nat$, $a > 0$ y $b > 0$. Si $(a-1) \ \% \ b = 0$ entonces $b - (a-1) \ \% \ b = b$ y podemos reemplazar el IF por el módulo de su resultado. Y en lugar de trabajar con $(a-1)$, aumentamos a en uno y nos deshacemos de la condición $a > 0$. La relación entonces la podemos ver en los siguientes lemas:

```
lemma divmod_div_mod_case2:
  assumes hyp: "divmod_rel(a,b,q,r)"
     and nat: "a ∈ Nat" "b ∈ Nat" "q ∈ Nat" "r ∈ Nat" and pos: "0 < b"
  shows "divmod_rel(-.(Succ[a]), b, -.(Succ[a ÷ b]), (b -- (Succ[a] % b)) % b)"
  apply (insert prems)
  unfolding divmod_rel_def div_def
  apply(cases "Succ[a] % b = 0" , simp_all)
proof (auto)
  assume h1: "Succ[q * b + r] % b = 0" and h2: "r < b"
  from nat have 1: "Succ[q * b + r] = q * b + Succ[r]" by simp
  from nat h2 have h2': "Succ[r] < Succ[b]" by auto
  from nat pos h1 1 have 2: "∃ q ∈ Nat : Succ[r] = q * b"
    using mod0_imp_dvd_nat[of "Succ[r]" b] by (simp del: add_Succ_nat)
  with nat pos h2' obtain q'
    where q': "q' ∈ Nat" "Succ[r] = q' * b" by auto
  with nat h2' pos have "q' = 1"
    by (auto elim!: natCases[of b] natCases[of q'], auto)
  with nat 1 q' show "Succ[q * b + r] = q * b + b" by auto
next
  assume h1: "Succ[q * b + r] % b ≠ 0" and h2: "r < b"
```

```
  from nat have 1: "Succ[q * b + r] = q * b + Succ[r]"  by simp
  from nat h2 have h2': "Succ[r] ≤ b"  by auto
  from nat pos h1 1 have "Succ[r] % b ≠ 0"  by (simp del: add_Succ_nat)
  with nat pos have "Succ[r] ≠ b"
    by (auto dest: mod0_imp_dvd_nat[of "Succ[r]" b])
  with nat pos h2' have 3: "Succ[r] % b = Succ[r]"
    by (simp add: nat_leq_less[of "Succ[r]" b]
      del: add_Succ_nat diff_add_assoc1 nat_Succ_leq_iff_less)
  show "-.(Succ[q * b + r]) = (b -- Succ[q * b + r] % b) % b + -.(q * b + b)"
    apply (insert nat pos h2')
    apply (rule minusInj)
    apply (simp add: 1 3 add_assoc_int[symmetric]
      del: add_Succ_nat nat_Succ_leq_iff_less diff_add_assoc1)
    by (cases "b = Succ[r]" , simp_all add: add_commute_nat)
next
  assume h1: "Succ[q * b + r] % b ≠ 0"  and h2: "r < b"
  from nat have 1: "Succ[q * b + r] = q * b + Succ[r]"  by simp
  from nat h2 have h2': "Succ[r] ≤ b"  by auto
  from nat pos h1 1 have "Succ[r] % b ≠ 0"  by (simp del: add_Succ_nat)
  with nat pos have "Succ[r] ≠ b"
    by (auto dest: mod0_imp_dvd_nat[of "Succ[r]" b])
  with nat pos h2' have 3: "Succ[r] % b = Succ[r]"
    by (simp add: nat_leq_less[of "Succ[r]" b]
      del: add_Succ_nat diff_add_assoc1 nat_Succ_leq_iff_less)
  from nat pos h1 h2 show "(b -- Succ[q * b + r] % b) % b < b"
    apply (simp add: 1 3 del: add_Succ_nat nat_Succ_leq_iff_less)
    by (cases "b = Succ[r]" , simp_all)
qed

lemma divmod_rel_div_mod_case2:
  assumes "divmod_rel(a,b,q,r)"  and "a ∈ Nat" "b ∈ Nat" "q ∈ Nat" "r ∈ Nat" "0 < b"
  shows "divmod(-.(Succ[a]), b) = ⟨ -.(Succ[a ÷ b]), (b -- (Succ[a] % b)) % b⟩ "

lemma div_neg_to_nat:        — De división de entero negativo a división de naturales.
  "⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(Succ[a]) ÷ b = -.(Succ[a ÷ b])"
  apply (rule div_unique, rule divmod_div_mod_case2, rule divmod_rel_div_mod[of a b])
  by auto

lemma mod_neg_to_nat:        — De módulo de entero negativo a módulo de naturales.
  "⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(Succ[a]) % b = (b -- (Succ[a] % b)) % b"
  apply (rule mod_unique, rule divmod_div_mod_case2, rule divmod_rel_div_mod[of a b])
  by auto
```

Propiedades sobre enteros negativos

Las mismas propiedades ya probadas para naturales son ahora probadas para enteros negativos. Para probarlas, simplemente tratamos de llevar las expresiones a naturales con *div_neg_to_nat* y *mod_neg_to_nat*, y usamos los lemas análogos ya probados.

```
lemma div_mult_self1_case2 [simp]:
  assumes q: "q ∈ Nat" and m: "m ∈ Nat" and n: "n ∈ Nat" and pos: "0 < n"
  shows " (-.(Succ[q]) + -.m * n) ÷ n = -.m + -.(Succ[q]) ÷ n" (is "?P(m)" )
using m proof (induct m)
  from assms show "?P(0)" by simp
next
  fix k
  assume k: "k ∈ Nat" and ih: "?P(k)"
  from q k n pos have 1: "-.(Succ[q] + Succ[k] * n) ÷ n = -.(Succ[(q + Succ[k] * n) ÷ n])"
    by (simp add: div_neg_to_nat)
  from q k n pos have 2: "-.(Succ[k]) + -.(Succ[q]) ÷ n = -.(Succ[k] + Succ[q ÷ n])"
    by (simp add: div_neg_to_nat)
  from q k n pos 1 2 show "?P(Succ[k])"
    by (simp del: add_Succ_left_nat mult_Succ_left_nat)
qed

lemma mod_mult_self1_case2 [simp]:
  assumes "q ∈ Nat" and "m ∈ Nat" and "n ∈ Nat" and "0 < n"
  shows " (-.(Succ[q]) + -.m * n)%n = (n -- (Succ[q]%n))%n"
  using prems mod_neg_to_nat[of "q + m * n" n]
  by (simp add: add_Succ_left_nat[symmetric] del: add_Succ_left_nat)
```

A partir de estos dos lemas se prueban fácilmente los siguientes:

```
lemma div_neg1 [simp]: "⟦ b ∈ Nat; 0 < b⟧ ⟹ -.1 ÷ b = -.1"
lemma mod_neg1 [simp]: "⟦ b ∈ Nat; 0 < b⟧ ⟹ -.1%b = b -- 1"
lemma div_neg_self [simp]: "⟦ b ∈ Nat; 0 < b⟧ ⟹ -.b ÷ b = -.1"
lemma mod_neg_self [simp]: "⟦ b ∈ Nat; 0 < b⟧ ⟹ -.b%b = 0"

lemma div_mult_neg_self1_is_id [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(a ∗ b) ÷ b = -.a"
lemma mod_mult_neg_self1_is_0 [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(a ∗ b)%b = 0"

lemma div_neg_by_1 [simp]: "m ∈ Nat ⟹ -.m ÷ 1 = -.m"
lemma mod_neg_by_1 [simp]: "m ∈ Nat ⟹ -.m%1 = 0"
```

Sobre la distributividad con la resta de naturales. Los dos lemas siguientes son los principales. Los que siguen se prueban a partir de estos.

```
lemma div_neg_diff_self1 [simp]:
  assumes "a ∈ Nat" "b ∈ Nat" "0 < b" "a < b"
  shows "-.(b -- a) ÷ b = -.1"
  apply (insert prems)
  apply (rule natCases[of a], auto simp add: diff_Succ_nat)
proof -
  fix x
  assume nat: "x ∈ Nat" "b ∈ Nat" and h: "Succ[x] < b" and pos: "0 < b"
  from nat h have 1: "Succ[Succ[x]] ≤ b"
    using nat_less_iff_Succ_leq[symmetric] by simp
  with nat have 2: "Pred[b -- x] = Succ[b -- Succ[Succ[x]]]"
```

```
   using pred_diff_to_Succ by simp
  from nat pos 1 show "-.(Pred[b -- x]) ÷  b = -.1"
   using div_neg_to_nat by (auto simp add: 2)
qed

lemma mod_neg_diff_self1 [simp]:
 "⟦ a ∈  Nat; b ∈  Nat; 0 < b; a < b⟧ ⟹  -.(b -- a)%b = a"
 apply (insert prems)
 apply (rule natCases[of a], auto simp add: diff_Succ_nat)
proof -
 fix x
 assume nat: "x ∈  Nat"  "b ∈  Nat"  and pos: "0 < b"  and h: "Succ[x] < b"
 from nat h have 1: "Succ[Succ[x]] ≤  b"
  using nat_less_iff_Succ_leq[symmetric] by simp
 from nat 1 pos have 2: "Succ[b -- Succ[Succ[x]]] < b"
  by (simp add: diff_Succ_nat[of b "Succ[x]" ] diff_Succ_less)
 from nat 1 have 3: "Pred[b -- x] = Succ[b -- Succ[Succ[x]]]"
  by (rule pred_diff_to_Succ)
 from nat pos 1 show "-.(Pred[b -- x])%b = Succ[x]"
  apply (simp add: 2 3 mod_neg_to_nat[of "b -- Succ[Succ[x]]"  b])
  using gt0_implies_Suc[of b] by auto
qed

lemma div_add_self_case2 [simp]: "⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (-.a + b) ÷ b = -.a ÷ b + 1"
lemma div_add_self_case3 [simp]: "⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a + -.b) ÷ b = a ÷ b + -.1"
lemma div_add_self1_case4 [simp]: "⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(a + b) ÷ b = -.a ÷ b + -.1"
lemma mod_add_self_case2 [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (-.a + b)%b = -.a%b"
lemma mod_add_self_case3 [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a + -.b)%b = a%b"
lemma mod_add_self1_case4 [simp]: "⟦ a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(a + b)%b = -.a%b"
```

Capítulo 7

Caso de ejemplo

El sistema de pruebas TLAPS cuenta con varios casos de ejemplo ya desarrollados. Los ejemplos, escritos en TLA^{+2}, son especificaciones de los sistemas o algoritmos junto con las pruebas de sus invariantes. Para testear la aritmética desarrollada en Isabelle/TLA^{+}, se tomó como caso de prueba uno de estos ejemplos, el llamado *Algoritmo de la Panadería*. Se lo eligió por varias razones: (i) utiliza aritmética, (ii) es uno de los casos de ejemplo más desarrollados y más complejos, y (iii) la estructura de su especificación es típica de las especificaciones reales de TLA^{+}. El test consistió en dos partes.

En primer lugar se tradujo la especificación del algoritmo de TLA^{+2} a Isabelle/TLA^{+}, y se probaron sus invariantes como teoremas de Isabelle. Esto ayudó a ajustar las reglas de Isabelle/TLA^{+} incluidas en el Simplificador y la configuración de los métodos automáticos.

La segunda parte del test consistió en comparar la performance del proceso de verificación del TLAPS sobre las pruebas del algoritmo. Primero usando definiciones y axiomas aritméticos ad-hoc, y luego reemplazándolos por sintaxis desarrollada en las nuevas teorías aritméticas de Isabelle/TLA^{+}.

A continuación, damos una introducción al algoritmo probado y su especificación formal en el lenguaje de algoritmos PlusCal y la traducción a Isabelle/TLA^{+}. Luego se detallan las pruebas de las invariantes (primer parte del test), y la comparación de las pruebas (segunda parte).

7.1. El Algoritmo de la Panadería

El *Algoritmo de la Panadería* [Lam74] es una de las soluciones más simples conocidas del problema de exclusión mutua entre *n* procesos. La idea es que cada proceso inactivo, es decir aquellos que no están en la zona crítica, tiene una variable (llamada *num* en el algoritmo) que indica la posición de ese proceso en una hipotética cola de todos los procesos inactivos. Cada proceso en la cola lee las variables de los otros procesos y entra a la sección crítica sólo después de determinar que es el primero de la cola. A diferencia de otros algoritmos similares, este tiene la propiedad que funciona incluso cuando una lectura de memoria que coincide con una escritura en memoria. Aquí se usará una versión simplificada del algoritmo de la panadería, en el sentido de que las lecturas y escrituras en memoria se consideran acciones atómicas.

La especificación formal está escrita en el lenguaje de algoritmos PlusCal [Lam09]. PlusCal es un lenguaje de algoritmos basado en TLA^{+}. No puede ser compilado en un código ejecutable eficiente, pero un algoritmo escrito en PlusCal puede traducirse a una especificación en TLA^{+} que puede verificarse en un model-checker o razonar sobre esta en cualquier nivel de formalidad.

En la figura 7.1 vemos el algoritmo en PlusCal. Se usa la notación de TLA^{+} y la semántica tradicional, pero haremos algunas aclaraciones con respecto a la sintaxis propia de PlusCal. El comando *process* define los procesos, donde *P* es el conjunto de procesos, y las variables *unread*, *max* y *nxt* son locales

al proceso. Habrá tantos procesos ejecutándose simultáneamente como elementos de P. La variable $self$ es el identificador propio de cada proceso, aquí $self \in P$. Las etiquetas $p_1, \ldots, p_8$ determinan los pasos del algoritmo. Una acción atómica consiste en la ejecución de una etiqueta a otra. El comando "with *(id* $\in S$*)* do *body*" setea *id* a un elemento elegido de manera no determinística del conjunto S y luego ejecuta *body*. El comando *await F* se ejecuta luego de que la fórmula F sea verdadera. El comando *skip* no hace nada.

```
−algorithm AtomicBakery {
variable num = [i ∈ P ↦ 0],
  flag = [i ∈ P ↦ FALSE];

process (p ∈ P)
  variables unread, max, nxt {
p1 : while (TRUE) {
       unread := P \ {self};
       max := 0;
       flag [self] := TRUE;

p2 :   while (unread ≠ {}) {
         with (i ∈ unread) {
           unread := unread \ {i};
           IF (num[i] > max) { max := num[i]; }
         }
       };
p3 :   num[self] := max + 1;
p4 :   flag[self] := FALSE;
       unread := P \ {self};
p5 :   while (unread ≠ {}) {
         with (i ∈ unread){ nxt := i; };
         await ¬flag[nxt];
p6 :     await ∨ num[nxt] = 0
          ∨ IF self > nxt
            THEN num[nxt] > num[self]
            ELSE num[nxt] ≥ num[self];
          unread := unread \ {nxt};
       };
p7 :   skip ; \\sección crítica
p8 :   num[self] := 0;
     }
   }
}
```

Figura 7.1: Algoritmo de la Panadería simplificado en lenguaje PlusCal

La traducción de PlusCal a TLA+ se hace automáticamente con la herramienta PlusCal Traslator. Aquí mostramos el resultado de la traducción adaptado a Isabelle/TLA+. Durante la adaptación se pierde parte del azúcar sintáctico de TLA+, como las variables y fórmulas primadas, la palabra clave *UNCHANGED*, los módulos, etc.

(En un primer momento, las variables de la especificación –también las correspondientes variables primadas– estaban declaradas como constantes de tipo c. Sin embargo, esto dificultaba las pruebas, por lo que ahora, aunque no se muestren para simplificar la lectura, las variables están pasadas como parámetros en cada definición hecha.)

```
consts P :: "c"      — Conjunto de procesos del sistema
  defaultInitValue :: "c"      — Variable creada durante la traducción de PlusCal

axioms
  PInNat: "P ⊆ Nat"      — Los procesos se identifican con números naturales

definition Init where
  "Init ≡ num = [i ∈ P ↦ 0]
        ∧ flag = [i ∈ P ↦ FALSE]
        ∧ unread = [self ∈ P ↦ defaultInitValue]
        ∧ max = [self ∈ P ↦ defaultInitValue]
        ∧ nxt = [self ∈ P ↦ defaultInitValue]
```

```
    ∧ pc = [self ∈ P ↦ ''p1'']"
```

Mostraremos como ejemplo sólo la traducción de la acción *p2*. Vemos que el identificador del proceso *pc[self] = "p2"*. Si se cumple la condición del *while*, la siguiente acción *pc'[self]* es otra vez *"p2"*; si no, pasa a la siguiente acción *"p3"*.

```
definition p2 where
  "p2 (self) ≡ pc[self] = ''p2''
            ∧ (IF unread[self] ≠ {}
               THEN ∃i ∈ unread[self] :
                  (  unread' = [unread EXCEPT ![self] = unread[self] \ {i}]
                   ∧ (IF num[i] > max[self]
                      THEN max' = [max EXCEPT ![self] = num[i]]
                      ELSE (max' = max)))
                   ∧ pc' = [pc EXCEPT ![self] = ''p2'']
               ELSE   pc' = [pc EXCEPT ![self] = ''p3'']
                  ∧ unread' = unread ∧ max' = max)
            ∧ num' = num ∧ flag' = flag ∧ nxt' = nxt"

definition p where
  "p(self) ≡ p1(self) ∨ p2(self) ∨ p3(self) ∨ p4(self) ∨ p5(self) ∨ p6(self) ∨ p7(self) ∨ p8(self)"

definition Next where
  "Next ≡ ∃ self ∈ P : p(self)"
```

7.2. Pruebas de las invariantes

Probaremos dos clases de invariantes sobre el algoritmo: la invariante de tipos, definida por la expresión llamada *TypeOK*, y la invariante general del sistema, llamada *Inv*.

```
definition TypeOK where
  "TypeOK ≡ num ∈ [P → Nat]
        ∧ flag ∈ [P → BOOLEAN]
        ∧ unread ∈ [P → SUBSET P ∪ { defaultInitValue} ]
        ∧ (∀ i ∈ P : pc[i] ∈ { ''p2'', ''p5'', ''p6''} ⇒ unread[i] ⊆ P ∧ i ∉ unread[i])
        ∧ max ∈ [P → Nat ∪ { defaultInitValue} ]
        ∧ (∀ j ∈ P : (pc[j] ∈ { ''p2'', ''p3''} ) ⇒ max[j] ∈ Nat)
        ∧ nxt ∈ [P → P ∪ { defaultInitValue} ]
        ∧ (∀ i ∈ P : (pc[i] = ''p6'') ⇒ nxt[i] ∈ unread[i] \ {i} )
        ∧ pc ∈ [P → { ''p1'',''p2'',''p3'',''p4'',''p5'',''p6'',''p7'',''p8''}]"

definition IInv where
  "IInv(i) ≡ ((num[i] = 0) = (pc[i] ∈ { ''p1'', ''p2'', ''p3''} ))
            ∧ (flag[i] = (pc[i] ∈ { ''p2'', ''p3'', ''p4''} ))
            ∧ (pc[i] ∈ { ''p5'', ''p6''} ⇒ (∀ j ∈ (P \ unread[i]) \ { i} : After(j,i)))
            ∧ ( pc[i] = ''p6''
                ∧ ( (pc[nxt[i]] = ''p2'') ∧ i ∉ unread[nxt[i]]
                    ∨ (pc[nxt[i]] = ''p3''))
```

```
                ⇒ max[nxt[i]] ≥ num[i])
  ∧ ((pc[i] ∈ { ''p7'', ''p8''} ) ⇒ (∀ j ∈ P \ { i} : After(j,i)))"

definition Inv where
  "Inv ≡ TypeOK ∧ (∀ i ∈ P : IInv(i))"
```

La especificación del sistema está dada por la fórmula $Spec \equiv Init \wedge \Box[Next]_{vars}$, pero todavía no es posible escribirla en Isabelle/TLA$^+$ ya que no están formalizados los operadores temporales. En su lugar, probaremos cada invariante sobre *Init* y *Next* por separado. Las invariantes sobre *Spec* se prueban trivialmente a partir de estas pruebas. Los siguientes son los teoremas que se deben probar, donde se ha expandido la definición de *Next*:

```
theorem InitType: Init ⟹ TypeOK                    — Invariante de tipos.
theorem NextType: ⟦ TypeOK; self ∈ P;  p(self) ⟧ ⟹ TypeOK'

theorem InitInv: Init ⟹ Inv                        — Invariante de general.
theorem NextInv: ⟦ Inv; self ∈ P;  p(self) ⟧ ⟹ Inv'
```

Las pruebas para *Init* son casi triviales. La estrategia para las pruebas de *Next* es la siguiente:

- En primer lugar, usando la definición de *p*, probar por separado el teorema para cada acción.
- Por cada acción, probar por separado los elementos de la conjunción que forma cada invariante.
- Si la fórmula de acción contiene alguna expresión (como IF o CASE) donde sea necesario hacer un análisis de casos (esto se da en las acciones *p*2, *p*5 y *p*6) entonces dividir el subobjetivo para cada caso.
- Cuando ya no haya casos que analizar en el subojetivo intentar probarlo con algún método automático. En general, alguna combinación de estos métodos da resultado. Si no, se necesitará probar algún paso intermedio antes.

Las pruebas completas junto con las definiciones del sistema se encuentran en el apéndice B. La cantidad de pasos de prueba de casos prueba de cada teorema puede darnos una idea de su tamaño:

InitType:	1 paso de prueba.
NextType:	6 pasos de prueba.
InitInv:	89 pasos de prueba.
NextInv:	1143 pasos de prueba.

Para dar una idea de la complejidad de la prueba, damos como ejemplo una parte de la prueba de *p*2 de la invariante general para los casos

$$unread[self] = \{\} \quad \text{y} \quad self = i$$

(El caso $unread[self] \neq \{\}$ es aún más complicado que este.) Esta prueba es equivalente a probar el lema:

```
lemma ⟦ p2(self); TypeOK; IInv(i); i ∈ P; self = i; unread[self] = {} ⟧ ⟹ IInv'(i)
```

A continuación se muestra el estado de la prueba luego de desarmar las definiciones de *p2_def*, *TypeOK_def* y *IInv_def* con el comando:

```
apply (simp add: p2_def TypeOK_def IInv_def)
```

La fórmula a probar es:

“⟦ self = i; num[i] = 0 ∧ boolify(flag[i]); i ∈ P; ∀ x : x ∉ unread[i];
pc[i] = ''p2'' ∧ (IF ∃x : x ∈ unread[i] THEN ∃ia ∈ unread[i] :
unread' = [unread EXCEPT ![i] = unread[i] \ {ia}]
∧ (IF max[i] < num[ia] THEN max' = [max EXCEPT [i] = num[ia]] ELSE max' = max)
∧ pc' = [pc EXCEPT [i] = ''p2''] ELSE pc' = [pc EXCEPT [i] = ''p3'']
∧ unread' = unread ∧ max' = max) ∧ num' = num ∧ flag' = flag ∧ nxt' = nxt;
num ∈ [P → Nat] ∧ flag ∈ [P → {FALSE, TRUE }]
∧ unread ∈ [P → addElt(defaultInitValue, SUBSET P)]
∧ (∀ i ∈ P : pc[i] = ''p2'' ∨ pc[i] = ''p5'' ∨ pc[i] = ''p6'' ⇒ unread[i] ⊆ P ∧ i ∉ unread[i])
∧ max ∈ [P → addElt(defaultInitValue, Nat)]
∧ (∀ j ∈ P : pc[j] = ''p2'' ∨ pc[j] = ''p3'' ⇒ max[j] ∈ Nat)
∧ nxt ∈ [P → addElt(defaultInitValue, P)]
∧ (∀ i ∈ P : pc[i] = ''p6'' ⇒ nxt[i] ∈ P ∧ nxt[i] ≠ i)
∧ pc ∈ [P → ''p1'', ''p2'', ''p3'', ''p4'', ''p5'', ''p6'', ''p7'', ''p8'']⟧
⟹ ([pc EXCEPT [i] = ''p3'']][i] = ''p1'' ∨ [pc EXCEPT ![i] = ''p3'']][i] = ''p2'' ∨
[pc EXCEPT ![i] = ''p3'']][i] = ''p3'')
∧ flag[i] = ([pc EXCEPT ![i] = ''p3'']][i] = ''p2'' ∨ [pc EXCEPT ![i] = ''p3'']][i] = ''p3'' ∨
[pc EXCEPT ![i] = ''p3'']][i] = ''p4'')
∧ ([pc EXCEPT ![i] = ''p3'']][i] = ''p5'' ∨ [pc EXCEPT ![i] = ''p3'']][i] = ''p6'' ⇒
(∀ j ∈ P \ unread[i] \ i : After(j, i, unread, max, flag, [pc EXCEPT ![i] = ''p3''], num, nxt)))”

En este caso, se termina la prueba con el método *clarsimp*.

7.3. Comparación en la performance de la certificación

En la segunda parte del caso de ejemplo, comparamos la performance en la certificación de dos versiones de la prueba de las invariantes del Algoritmo de la Panadería en TLAPS. Las pruebas en TLA^{+2} usadas en este trabajo están estructuradas de manera similar a la prueba del algoritmo en Isabelle/TLA$^+$ ya vista.

En la primera versión, las pruebas en TLA^{+2} ya estaban realizadas antes de finalizar la codificación de la aritmética en Isabelle/TLA$^+$, por lo que los operadores aritméticos necesarios y algunas de sus propiedades están definidos como axiomas. Así es que se agrega lo siguiente al comienzo de la especificación del algoritmo en TLA^{+2}:

```
CONSTANTS P, GT(_,_), Plus(_,_), Nat
ASSUME PsubsetNat ≡ P ⊆ Nat
GEQ(a, b) ≡ (a = b) ∨ GT(a, b)
```

donde P es el conjunto de identificadores de los procesos y se asume que los identificadores son números naturales (por el axioma PsubsetNat). La constante GT representa a la relación “mayor que”, GEQ a la relación “mayor o igual que”, Plus a la suma de naturales y Nat al conjunto de números naturales.

También deben agregarse ciertos teoremas que expresen las propiedades de las constantes declaradas (la mayoría de los cuales no pueden probarse). Los siguientes teoremas son usados en las pruebas de las invariantes:

```
THEOREM GTAxiom ≡ ∀ n, m ∈ Nat : ¬ (GT(n,m) ∧ GT(m,n))
THEOREM GEQAxiom ≡ ∀ n, m ∈ Nat : (n = m) ∨ GT(n, m) ∨ GT(m, n)
THEOREM GEQTransitive ≡ ∀ n, m, q ∈ Nat : GEQ(n,m) ∧ GEQ(m,q) ⇒ GEQ(n,q)
THEOREM Transitivity2 ≡ ∀ n, m, q ∈ Nat : GT(n,m) ∧ GEQ(m,q) ⇒ GT(n,q)
THEOREM GEQorLT ≡ ∀ n, m ∈ Nat : GEQ(n,m) ⇔ ¬GT(m, n)
  BY GEQAxiom, GTAxiom DEF GEQ
THEOREM ZeroANat ≡ 0 ∈ Nat
```

THEOREM Plus1 ≡ ∧ ∀ n ∈ Nat : Plus(1, n) ∈ Nat ∧ ∀ n ∈ Nat : GT(Plus(1, n), n)
THEOREM Plus1NZ ≡ ∀ n ∈ Nat : Plus(1, n) ≠ 0
THEOREM NatGEQZero ≡ ∀ n ∈ Nat: (n ≠ 0) ⇔ GT(n, 0)
THEOREM Plus1GT ≡ ∀ m, n ∈ Nat : GEQ(m, n) ⇒ GT(Plus(1,m), n)
BY Plus1, Transitivity2
THEOREM StringAssump ≡ ∧ ”p1”≠ ”p2”∧ ”p1”≠ ”p3”∧ ”p1”≠ ”p4”∧ ”p1”≠ ”p5”
∧ ”p1”≠ ”p6”∧ ”p1”≠ ”p7”∧ ”p1”≠ ”p8”∧ ”p1”≠ ”Done”∧ ”p2”≠ ”p3”∧ ”p2”≠ ”p4”
∧ ”p2”≠ ”p5”∧ ”p2”≠ ”p6”∧ ”p2”≠ ”p7”∧ ”p2”≠ ”p8”∧ ”p2”≠ ”Done”∧ ”p3”≠ ”p4”
∧ ”p3”≠ ”p5”∧ ”p3”≠ ”p6”∧ ”p3”≠ ”p7”∧ ”p3”≠ ”p8”∧ ”p3”≠ ”Done”∧ ”p4”≠ ”p5”
∧ ”p4”≠ ”p6”∧ ”p4”≠ ”p7”∧ ”p4”≠ ”p8”∧ ”p4”≠ ”Done”∧ ”p5”≠ ”p6”∧ ”p5”≠ ”p7”
∧ ”p5”≠ ”p8”∧ ”p5”≠ ”Done”∧ ”p6”≠ ”p7”∧ ”p6”≠ ”p8”∧ ”p6”≠ ”Done”∧ ”p7”≠ ”p8”
∧ ”p7”≠ ”Done”∧ ”p8”≠ ”Done”

El teorema StringAssump se agrega debido a la falta de la teoría sobre strings al momento de la prueba. Como puede notarse, estas propiedades corresponden a algunos de los lemas ya declarados y probados en las teorías de aritmética y de orden de los números naturales en Isabelle/TLA$^+$.

En la segunda versión, una vez formalizada la aritmética en Isabelle/TLA$^+$, se quitaron los axiomas aritméticos ad-hoc de la especificación. Además, en esta instancia, Zenon fue extendido para soportar los operadores aritméticos de TLA$^+$. Las constantes y teoremas anteriores fueron reemplazados por la sintaxis stándard de TLA$^+$ y por las decalraciones:

EXTENDS Naturals
CONSTANT P
ASSUME PsubsetNat ≡ P ⊆ Nat

donde se mantiene el conjunto de procesos P.

Se tomaron en cuenta los siguientes criterios de comparación:

- la cantidad de obligaciones de prueba generadas,
- el tiempo total que tarda el Administrador de Pruebas TLAPM en certificar la prueba (el proceso completo), y
- la cantidad de fallos de Zenon, es decir, las obligaciones de prueba que Zenon no pudo probar en el tiempo límite preestablecido de 10 segundos.

En la figura 7.2 podemos ver los resultados obtenidos. (Las pruebas se realizaron sobre TLAPM versión 0.9.2010_04_06_03, Isabelle versión 2009-1, Zenon versión 0.6.2, en una máquina Intel Core 2 Duo 2GHz con 2Gb de RAM.)

De la tabla de comparación obtuvimos las siguientes conclusiones:

- En la segunda versión hubo menos obligaciones generadas debido a que las pruebas en TLA^{+2} son más cortas: el mayor poder del razonamiento sobre aritmética de Isabelle/TLA$^+$ permite terminar las subpruebas en menor cantidad de niveles.

- En la comparación se incluyen los fallos de Zenon, es decir, las obligaciones de prueba que Zenon no pudo probar en el tiempo preestablecido de 10 segundos. De todas maneras, las obligaciones fueron igualmente probadas en Isabelle/TLA$^+$ (lo que suma más tiempo al total). Cada fallo significan 10 segundos extras en el tiempo total, por lo que los 143 fallos del segundo caso significan 23 min 50 seg. del tiempo total que Zenon estuvo intentando probar en vano.

- En la segunda versión no hay fallos de Zenon debido que Zenon no soporta los operadores aritméticos de la primera versión pero sí los operadores estándar de TLA$^+$. Esto influye de manera considerable en el tiempo total de certificación por lo explicado en el punto anterior. Sin embargo, debe tenerse en cuenta que las obligaciones se prueban en Zenon de todas formas (en menos de

	Con axiomas ad-hoc en TLA^{+2}	Sin axiomas ad-hoc y con teorías aritméticas en Isabelle/TLA$^+$
Invariante de tipos: $Init \wedge \Box[Next]_{vars} \Rightarrow \Box TypeOK$		
Líneas de prueba	290	92
Obligaciones generadas	79	24
Fallos de Zenon	9	0
Tiempo de certificación	5 min. 12 seg.	2 min. 42 seg.
Invariante del sistema: $Init \wedge \Box[Next]_{vars} \Rightarrow \Box Inv$		
Líneas de prueba	1565	557
Obligaciones generadas	533	188
Fallos de Zenon	143	0
Tiempo de certificación	35 min. 35 seg.	5 min. 38 seg.

Figura 7.2: Comparación de la performance en la certificación del Algoritmo de la Panadería.

10 segundos cada una) por lo que también lleva su tiempo y que además hay menos obligaciones a probar.

- Teniendo en cuenta las consideraciones mencionadas anteriormente, el tiempo total de certificación bajó considerablemente en la segunda versión en ambos casos. A pesar de los pocos operadores y axiomas ad-hoc agregados en la primera versión, el razonamiento sobre aritmética tuvo una gran influencia en la prueba. Esto también se nota en que la cantidad de líneas de prueba y de obligaciones generadas es mayor que en la segunda versión.

Capítulo 8

Conclusiones

8.1. Conclusiones

En este trabajo se extendió la lógica-objeto Isabelle/TLA$^+$, la axiomatización del lenguaje de especificación TLA$^+$ en el asistente de pruebas Isabelle. En particular, a las teorías existentes (lógica proposicional y de primer orden, teoría de conjuntos, funciones, puntos fijos y números naturales) se agregó el soporte para los operadores aritméticos estándar de TLA$^+$: suma, multiplicación, resta, exponenciación (definidos a partir de un esquema de recursión primitiva), división y módulo (definidos conjuntamente a través del Algoritmo de División).

Debido a que los operadores aritméticos tienen la misma sintaxis para los números naturales y para los enteros, la implementación fue incremental. En primer lugar, se definieron y probaron las propiedades de los operadores sobre números naturales (excepto la división y el módulo). Una vez hecho esto, se axiomatizó la noción de número entero negativo con la que se definió el conjunto de los enteros, como una extensión del conjunto de los naturales, y se formalizaron los operadores aritméticos para los casos todavía no definidos, es decir, cuando sus argumentos son números enteros negativos. En última instancia, se desarrollaron conjuntamente los operadores de división y módulo para naturales y enteros, ya que están intrínsecamente definidos por una función característica. Este desarrollo también llevó a extender significativamente la teoría sobre las relaciones de orden de los naturales y enteros.

Los operadores o relaciones sobre los enteros, están definidos en función del operador o relación correspondiente para los naturales. Por lo tanto el razonamiento aritmético sobre los enteros consiste principalmente en reducir las pruebas al razonamiento sobre los naturales. Así es que la mayor parte de la extensión de Isabelle/TLA$^+$ se llevó a cabo en la teoría sobre aritmética de naturales.

En particular, uno de los resultados que se llegó a probar es que el conjunto *Int* junto a los operadores $+$ y $*$ satisfacen las propiedades de un dominio entero (y, por extensión, de un anillo conmutativo). Entre otros resultados probados se incluyen la monotonía de la suma y multiplicación, $(Nat, \leq)$ como grupo parcialmente ordenado, formas de asociatividad y cancelación de la resta, y el ya mencionado Algoritmo de División para enteros.

En total, se han agregado a Isabelle/TLA$^+$ 17 axiomas (16 de los cuales son sobre enteros negativos, el restante es el esquema de recursión) y se han probado 507 teoremas sobre los operadores definidos: 185 correspondientes a la teoría sobre aritmética de los naturales, 126 a la de enteros, 101 a la teoría sobre división y 95 teoremas que se agregaron a la teoría de órdenes entre naturales.

Finalmente, se desarrolló un caso de ejemplo en el que se tradujo el Algoritmo de la Panadería, cuya especificación se encontraba en TLA$^+$, a Isabelle/TLA$^+$ y se probaron sus invariantes (de tipo y general del sistema), lo que contribuyó a testear la extensión realizada. Por otra parte, se analizó la comparación de la performance en la certificación del algoritmo en el Administrador de Pruebas TLAPS, con las

versiones de Isabelle/TLA$^+$ antes y después de añadir las teorías aquí definidas, y cuyas conclusiones ya fueron expuestas. Esto contribuyó a validar la interacción entre el Administrador de Pruebas TLAPM e Isabelle/TLA$^+$.

Uno de los objetivos de este trabajo fue aumentar el poder de razonamiento de los métodos de prueba automáticos de Isabelle/TLA$^+$ (y, por extensión, la certificación del Sistema de Pruebas TLAPS), por lo que fue importante la tarea de decidir cuáles de los teoremas probados debían ser añadidos por defecto al Simplificador o a cuáles se los debía agregar al Razonador Clásico como regla de introducción, de eliminación o de destrucción. Una asignación errónea puede hacer que en las pruebas subsiguientes los métodos fallen o, peor, que no terminen. En un principio se pensaba que un algoritmo de bajo nivel (como el algoritmo de Cooper, ver más abajo) debía ser implementado para mejorar el rendimiento de las pruebas (es decir, para automatizarlas más y para que sean más rápidas). Aunque un algoritmo de este tipo ayudaría, a partir del desarrollo del Algoritmo de División y de los resultados del caso de ejemplo, creemos que los resultados son satisfactorios.

Podemos mencionar los siguientes problemas encontrados durante la formalización. Al no haber inferencia de tipos, todos los teoremas declarados tienen como premisas que las variables pertenezcan a *Nat* o *Int*. Esto dificulta la tarea de los métodos automáticos, que fallan al intentar unificar esas premisas con la clausura de las variables sobre las operaciones. Por ejemplo, el teorema sobre la propiedad distributiva de los naturales sobre + y $*$:

theorem *"⟦ m ∈ Nat; n ∈ Nat; p ∈ Nat ⟧ ⟹ m * (n + p) = m * n + m * p"*

podría probarse simplemente agregando las reglas sobre conmutatividad y asociatividad a la prueba por inducción:

by *(induct set: Nat, auto simp add: add_commute_nat add_assoc_nat)*

Pero los métodos *auto* y *simp* fallan al unificar, por ejemplo, las expresiones $n+p$ o $m*n$ con las premisas de las reglas agregadas. En cambio, la prueba debe desarrollarse en detalle, paso por paso. Esto no constituye un error, pero es incómodo, y una solución facilitaría mucho los pruebas.

Otro de los detalles que se necesita corregir es la representación tradicional de los números. En este momento cada número tiene una traducción explícita (por ejemplo, 2 es Succ[Succ[0]]) y están representados hasta el número 15. Lo que hace falta es una representación automática implementada a bajo nivel. De esto además se desprende el problema del cálculo de expresiones con números altos. Por ejemplo, el siguiente lema, que no incluye siquiera cuantificadores o variables, tarda alrededor de un segundo en ser probado, tiempo considerable teniendo en cuenta su simplicidad.

lemma *"1 + 14 = 15"* **by** *simp*

8.2. Trabajos Futuros

Para completar los operadores estándar de TLA$^+$ sobre números falta la definición y aritmética de los números reales. El módulo estándar *Reals* de TLA$^+$ define los números reales y sus operadores aritméticos (los mismos aquí definidos para naturales y enteros).

De la misma manera que se derivaron los números naturales de los axiomas de Peano, se puede declarar como axioma que los reales junto con las operaciones de suma y multiplicación y la relación "menor que" forman un campo ordenado completo, y obtener de allí el conjunto *Reals* y sus operaciones y relaciones de orden. Otra opción es construirlos usando cortes de Dedekind [Lan51], con lo que antes se necesitaría la definición de los números racionales. Cualquiera sea el caso, es necesario hacer un análisis previo completo.

En cuanto a la aritmética de los naturales y enteros, hay todavía muchas propiedades más avanzadas por agregar y teorías por extender. Por ejemplo, falta desarrollar las propiedades de la exponenciación y la división de los naturales y los enteros. Hay que tener en cuenta que el objetivo final no es desarrollar una teoría matemática avanzada, sino verificar propiedades de sistemas concurrentes (principalmente). El nivel de sofisticación de estas teorías aritméticas dependerá del análisis de los resultados de casos de ejemplo y casos reales que se lleven a cabo en el futuro.

La aritmética de Presburger es una lógica de primer orden sobre enteros con suma y la relación "menor que". Es más débil que la teoría de Peano pero, a diferencia de ésta, es decidible: existe un algoritmo que decide si una fórmula de Presburger es verdadera. Uno de estos algoritmos es el de Cooper [Coo72] y ya está implementado en ML para Isabelle/HOL. Una instancia de este algoritmo para Isabelle/TLA$^+$ puede usarse como un nuevo método de prueba automático para este tipo de fórmulas, lo que facilitaría las pruebas y el razonamiento sobre estas. También se pueden implementar otros *decision procedures* en ML, como los que ya existen para Isabelle/HOL. Por ejemplo, hay varios algoritmos sobre cancelación de términos en fórmulas como

$$A + n * (m \div n) + B + (m \,\%\, n) + C \equiv A + B + C + m.$$

Por último, para completar la formalización de la sintaxis de TLA$^+$ faltan otros operadores, símbolos y expresiones no-aritméticas. Principalmente los operadores temporales ($\Box F$, $\Diamond F$, $WF_e(\mathcal{A})$, SF, $F \rightsquigarrow G$). No es necesario formalizar los operadores de acción $'$ (priming), $[A]_e$, $\langle A \rangle_e$, ENABLED, UNCHANGED, $A \cdot B$ (composición de acciones) porque, como ya se explicó, sus definiciones son expandidas y llevadas al nivel de las constantes. También falta desarrollar el módulo estándar *Bags* con sus operadores $\oplus$ y $\ominus$.

Bibliografía

[AL94] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Trans. Program. Lang. Syst.*, 16(5):1543–1571, 1994.

[BCBdODF09] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. *Automated Deduction –CADE-22*, pages 151–156, 2009.

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *Proc. 14th LPAR*, pages 151–165, 2007.

[CDLM08] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. A TLA$^+$ Proof System. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proc. LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 17–37, 2008.

[CDLM10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying Safety Properties With the TLA$^+$ Proof System. In J. Giesl and R. Hähnle, editors, *5th Intl. Joint Conf. Automated Reasoning (IJCAR 2010)*, Lecture Notes in Computer Science. Springer, 2010. To appear.

[CN08] Amine Chaieb and Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *Journal of Automated Reasoning*, 41(1):33–59, 07 2008.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press, 1972.

[Cor94] Intel Corporation. Statistical analysis of floating point flaw. Disponible en `http://support.intel.com/support/processors/pentium/fdiv/wp/`, November 1994.

[For03] U.S.-Canada Power System Outage Task Force. Final report on the August 14th, 2003 blackout in the United States and Canada. Disponible en `https://reports.energy.gov/`, November 2003.

[Hue75] G. P. Huet. A unification algorithm for typed λ-calculus. In *Theoretical Computer Science*, volume 1, pages 27–57, 1975.

[isa] Isabelle Home Page. `http://isabelle.in.tum.de`.

[Kal95] Sara Kalvala. A formulation of TLA in Isabelle. In *TPHOLs*, pages 214–228, 1995.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, agosto 1974.

[Lam93] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August 1993.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam03] Leslie Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.

[Lam09] L. Lamport. The PlusCal Algorithm Language. In Martin Leucker and Carroll Morgan, editors,

Theoretical Aspects of Computing-ICTAC, number 5684 in Lecture Notes in Computer Science, pages 36–60, 2009.

[Lan51] Edmund Landau. *Foundations of Analysis. The Arithmetic of Whole, Rational, Irrational and Complex Numbers.* Translated by F. Steinhardt. Chelsea Publishing Company, New York, N.Y., 1951.

[Lei69] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach, New York, 1969.

[Mer97] Stephan Merz. Yet another encoding of TLA in Isabelle. Technical report, Institut für Informatik, TU München, 1997.

[Mer08] Stephan Merz. *Logics of Specification Languages*, chapter The Specification Language TLA$^+$, pages 401–451. Springer Monographs in Theoretical Computer Science, 2008.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[Nip] Tobias Nipkow. A tutorial introduction to structured isar proofs. Disponible en `http://isabelle.in.tum.de/dist/Isabelle/doc/isar-overview.pdf`.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. In *Logic Programming*, volume 3, pages 237–258, 1986.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Automated Reasoning*, 5:363–397, 1989.

[Pau04] Lawrence C. Paulson. Introduction to isabelle. Technical report, Computer Laboratory, University of Cambridge, 2004.

[Pau06] Lawrence C. Paulson. Defining functions on equivalence classes. *ACM Trans. Comput. Logic*, 7(4):658–675, 2006.

[Pro96] Prof J. L. Lions and Mr Remy Hergott (cnes and Mr Bernard Humbert (aerospatiale and Mr Eric Lefort (esa. ARIANE 5, Flight 501 Failure, Report by the Inquiry Board. European Space Agency, 1996.

[tlaa] TLA$^+$ Proof System web site. `http://msr-inria.inria.fr/~doligez/tlaps/`.

[tlab] TLA$^+$ web site. `http://www.tlaplus.net`.

[Typ08] The Coq Development Team (Project TypiCal). The Coq proof assistant reference manual, 2008. `http://coq.inria.fr/V8.1pl3/refman/index.html`.

[Wen09] Makarius Wenzel. The Isabelle/Isar reference manual, December 2009. Disponible en `http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/isar-ref.pdf`.

Apéndice A

Teorías ariméticas de Isabelle/TLA$^+$ sobre los números naturales y enteros

En este apéndice se encuentra el complemento a las secciones desarrollas de Isabelle/TLA$^+$. En la teoría *Peano* sólo se muestra la parte en este desarrollada en este trabajo.

A.1. Peano's axioms and natural numbers

theory *Peano*
imports *FixedPoints Functions*
begin

Primitive Recursive Functions

We axiomatize a primitive recursive scheme for functions with one argument and domain on natural numbers. Later, we use it to define addition, multiplication, difference and exponentiation.

axioms
primrec_nat: "∃f : isAFcn(f) ∧ DOMAIN f = Nat
∧ f[0] = e ∧ (∀ n ∈ Nat : f[Succ[n]] = h(n,f[n]))"

lemma *bprimrec_nat:*
assumes *e: "e ∈ S"* **and** *suc: "∀ n ∈ Nat : ∀ x ∈ S : h(n,x) ∈ S"*
shows *"∃f ∈ [Nat → S] : f[0] = e ∧ (∀ n ∈ Nat: f[Succ[n]] = h(n,f[n]))"*
proof -
from *primrec_nat[of e h]* **obtain** *f* **where**
1: "isAFcn(f)" **and** *2: "DOMAIN f = Nat"*
and *3: "f[0] = e"* **and** *4: "∀ n∈Nat : f[Succ[n]] = h(n,f[n])"*
by *blast*
have *"∀ n∈Nat : f[n] ∈ S"*
proof *(rule natInduct)*
from *3 e* **show** *"f[0] ∈ S"* **by** *simp*
next
fix *n* **assume** *"n ∈ Nat"* **and** *"f[n] ∈ S"*
with *suc 4* **show** *"f[Succ[n]] ∈ S"* **by** *force*

qed
with *1 2 3 4* **show** *?thesis* **by** *blast*
qed

end

A.2. Orders on natural numbers

theory *NatOrderings*
imports *Peano*
begin

Using the sets *upto* we can now define the standard ordering on natural numbers. The constant ≤ is defined over the naturals by the axiom (conditional definition) *nat_leq_def* below; it should be defined over other domains as appropriate later on.

We generally define the constant < such that $a<b$ iff $a\leq b \wedge a\neq b$, over any domain.

definition
leq :: "[c,c] ⇒ c" (**infixl** *"<="* *50*)
where
nat_leq_def: "⟦m ∈ Nat; n ∈ Nat⟧ ⟹ (m <= n) ≡ (m ∈ upto(n))"

abbreviation *(input)*
geq :: "[c,c] ⇒ c" (**infixl** *">="* *50*)
where
"x >= y ≡ y <= x"

notation *(xsymbols)*
leq (**infixl** *"≤"* *50*) **and**
geq (**infixl** *"≥"* *50*)

Operator definitions and generic facts about <

definition
less :: "[c,c] ⇒ c" (**infixl** *"<"* *50*)
where
"a < b ≡ a ≤ b ∧ a ≠ b"

abbreviation *(input)*
greater :: "[c,c] ⇒ c" (**infixl** *">"* *50*)
where
"x > y ≡ y < x"

lemma *boolify_less [simp]: "boolify(a < b) = (a < b)"*
by *(simp add: less_def)*

lemma *less_isBool [intro!,simp]: "isBool(a<b)"*
by *(simp add: less_def)*

lemma *less_imp_leq [elim]: "a < b ⟹ a ≤ b"*
unfolding *less_def* **by** *simp*

lemma *less_irrefl [simp]: "(a < a) = FALSE"*
unfolding *less_def* **by** *simp*

lemma *less_irreflE [elim!]: "a < a ⟹ R"*
by *simp*

lemma *less_not_refl: "a < b ⟹ a ≠ b"*
by *auto*

lemma *neq_leq_trans [trans]: "a ≠ b ⟹ a ≤ b ⟹ a < b"*
by *(simp add: less_def)*

lemma *leq_neq_trans [trans,elim]: "a ≤ b ⟹ a ≠ b ⟹ a < b"*
by *(simp add: less_def)*

lemma *leq_neq_trans' [trans,elim]: "a ≤ b ⟹ b ≠ a ⟹ a < b"*
by *(simp add: less_def)*

lemma *leq_neq_iff_less: "a ≤ b ⟹ (a ≠ b) = (a < b)"*
by *auto*

Facts about ≤ over *Nat*

lemma *nat_boolify_leq [simp]: "⟦m ∈ Nat; n ∈ Nat⟧ ⟹ boolify(m ≤ n) = (m ≤ n)"*
by *(simp add: nat_leq_def)*

lemma *nat_leq_isBool [intro,simp]: "⟦m ∈ Nat; n ∈ Nat⟧ ⟹ isBool(m ≤ n)"*
by *(simp add: nat_leq_def)*

lemma *nat_leq_refl [intro,simp]: "n ∈ Nat ⟹ n ≤ n"*
unfolding *nat_leq_def* **by** *(rule uptoRefl)*

lemma *eq_leq_bothE:* — reduce equality over integers to double inequality
"⟦m ∈ Nat; n ∈ Nat; m = n; ⟦ m ≤ n; n ≤ m ⟧ ⟹ P⟧ ⟹ P" **by** *simp*

lemma *nat_zero_leq [simp]: "n ∈ Nat ⟹ 0 ≤ n"*
unfolding *nat_leq_def[OF zeroIsNat]* **by** *(rule zeroInUpto)*

lemma *nat_leq_zero [simp]: "n ∈ Nat ⟹ (n ≤ 0) = (n = 0)"*
by *(simp add: nat_leq_def uptoZero)*

lemma *nat_leq_SuccI [elim!,simp]:*
"⟦m ≤ n; m ∈ Nat; n ∈ Nat⟧ ⟹ m ≤ Succ*[n]"*
by *(auto simp add: nat_leq_def uptoSucc)*

lemma *nat_leq_Succ:*
"⟦m ∈ Nat; n ∈ Nat⟧ ⟹ (m ≤ Succ[n]) = (m ≤ n ∨ m = Succ[n])"
by *(auto simp add: nat_leq_def uptoSucc)*

lemma *nat_leq_SuccE [elim]:*
"⟦m ≤ Succ[n]; m ∈ Nat; n ∈ Nat; m ≤ n ⟹ P; m = Succ[n] ⟹ P⟧ ⟹ P"
by *(auto simp add: nat_leq_Succ)*

lemma *nat_leq_limit:*
"⟦m ≤ n; ¬(Succ[m] ≤ n); m ∈ Nat; n ∈ Nat⟧ ⟹ m = n"
by *(auto simp add: nat_leq_def intro: uptoLimit)*

lemma *nat_leq_trans [trans]:*
"⟦k ≤ m; m ≤ n; k ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ k ≤ n"
by *(auto simp add: nat_leq_def elim: uptoTrans)*

lemma *nat_leq_antisym :*
"⟦m ≤ n; n ≤ m; m ∈ Nat; n ∈ Nat⟧ ⟹ m = n"
by *(auto simp add: nat_leq_def elim: uptoAntisym)*

lemma *nat_Succ_not_leq_self [simp]:*
"n ∈ Nat ⟹ (Succ[n] ≤ n) = FALSE"
by *(auto dest: nat_leq_antisym)*

lemma *nat_Succ_leqD:*
"⟦Succ[m] ≤ n; m ∈ Nat; n ∈ Nat⟧ ⟹ m ≤ n"
apply *(subgoal_tac "m ≤ Succ[m]")*
by *(elim nat_leq_trans, auto)*

lemma *nat_Succ_leq_Succ :*
"⟦m ∈ Nat; n ∈ Nat⟧ ⟹ (Succ[m] ≤ Succ[n]) = (m ≤ n)"
by *(auto simp add: nat_leq_Succ intro: nat_leq_limit elim: nat_Succ_leqD)*

lemma *nat_leq_linear: "⟦m ∈ Nat; n ∈ Nat⟧ ⟹ m ≤ n ∨ n ≤ m"*
unfolding *nat_leq_def* **using** *uptoLinear* **.**

lemma *nat_leq_cases [case_names leq greater]:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
and *leq: "m ≤ n ⟹ P"* **and** *geq: "⟦n ≤ m; n ≠ m⟧ ⟹ P"*
shows *"P"*
proof *(cases "m ≤ n")*
case *True* **thus** *"P"* **by** *(rule leq)*
next
case *False*
with *m n* **have** *nm: "n ≤ m"* **by** *(blast dest: nat_leq_linear)*
thus *"P"*
proof *(cases "n=m")*

```
    case True
    with m have "m ≤ n" by simp
    thus "P" by (rule leq)
  next
    case False
    with nm show "P" by (rule geq)
  qed
qed

lemma nat_leq_induct: — sometimes called "complete induction"
  assumes "P(0)" and "∀n ∈ Nat : (∀m ∈ Nat : m ≤ n ⇒ P(m)) ⇒ P(Succ[n])"
  shows "∀n ∈ Nat : P(n)"
proof -
  from prems have "∀n ∈ Nat : ∀m ∈ Nat : m ≤ n ⇒ P(m)"
    by (intro natInduct, auto simp add: nat_leq_Succ)
  thus ?thesis by (blast dest: nat_leq_refl)
qed

lemma nat_leq_inductE:
  "⟦n ∈ Nat; P(0); ⋀n. ⟦n ∈ Nat; ∀m ∈ Nat : m ≤ n ⇒ P(m)⟧ ⟹ P(Succ[n])⟧ ⟹ P(n)"
  by (blast dest: nat_leq_induct)
```

Facts about $<$ over *Nat*

```
lemma nat_Succ_leq_iff_less [simp]:
  "⟦m ∈ Nat; n ∈ Nat⟧ ⟹ (Succ[m] ≤ n) = (m < n)"
  by (auto simp add: less_def dest: nat_Succ_leqD nat_leq_limit)

— alternative definition of < over Nat
lemmas nat_less_iff_Succ_leq = sym[OF nat_Succ_leq_iff_less, standard]
```

Reduce $\leq$ to $<$.

```
lemma nat_leq_less: — premises needed for isBool(m≤n) and reflexivity
  "⟦m ∈ Nat; n ∈ Nat⟧ ⟹ m ≤ n = (m < n ∨ m = n)"
  by (auto simp add: less_def)

lemma nat_less_Succ_iff_leq [simp]:
  "⟦m ∈ Nat; n ∈ Nat⟧ ⟹ (m < Succ[n]) = (m ≤ n)"
  by (simp del: nat_Succ_leq_iff_less add: nat_less_iff_Succ_leq nat_Succ_leq_Succ)

lemmas nat_leq_iff_less_Succ = sym[OF nat_less_Succ_iff_leq, standard]

lemma nat_not_leq_one [simp]:
  "n ∈ Nat ⟹ (¬(1 ≤ n)) = (n = 0)"
  by (cases set: Nat, auto)
```

$<$ and *Succ*.

```
lemma nat_not_less0 [simp]:
```

"n ∈ Nat ⟹ (n < 0) = FALSE"
by *(auto simp add: less_def)*

lemma *nat_less_SuccI:*
"⟦m < n; m ∈ Nat; n ∈ Nat⟧ ⟹ m < Succ[n]" **by** *auto*

lemma *nat_Succ_lessD:*
"⟦Succ[m] < n; m ∈ Nat; n ∈ Nat⟧ ⟹ m < n" **by** *(simp add: less_def)*

lemma *nat_less_leq_not_leq:*
"⟦m ∈ Nat; n ∈ Nat⟧ ⟹ (m < n) = (m ≤ n ∧ ¬ n ≤ m)"
by *(auto simp add: less_def dest: nat_leq_antisym)*

lemma *nat_Succ_not_less_self [simp]:*
"n ∈ Nat ⟹ Succ[n] < n = FALSE"
by *(simp add: less_def)*

Transitivity.

lemma *nat_less_trans :*
"⟦k < m; m < n; k ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ k < n"
by *(auto simp add: less_def dest: nat_leq_trans nat_leq_antisym)*

lemma *nat_less_trans_Succ [trans]:*
assumes *lt1: "i < j"* **and** *lt2: "j < k"* **and** *nat: "i ∈ Nat" "j ∈ Nat" "k ∈ Nat"*
shows *"Succ[i] < k"*
proof -
from *lt1 nat* **have** *"Succ[Succ[i]] ≤ Succ[j]"* **by** *simp*
also from *lt2 nat* **have** *"Succ[j] ≤ k"* **by** *simp*
finally show *?thesis* **using** *nat* **by** *simp*
qed

lemma *nat_leq_less_trans [trans]:*
"⟦k ≤ m; m < n; k ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ k < n"
by *(auto simp add: less_def dest: nat_leq_trans nat_leq_antisym)*

lemma *nat_less_leq_trans [trans]:*
"⟦k < m; m ≤ n; k ∈ Nat; m ∈ Nat; n ∈ Nat⟧ ⟹ k < n"
by *(auto simp add: less_def dest: nat_leq_trans nat_leq_antisym)*

Asymmetry.

lemma *nat_less_not_sym:*
"⟦m < n; m ∈ Nat; n ∈ Nat⟧ ⟹ ¬ (n < m)"
by *(simp add: nat_less_leq_not_leq)*

lemma *nat_less_asym:*
"⟦(¬ P ⟹ n < m); m < n; m ∈ Nat; n ∈ Nat⟧ ⟹ P"
by *(blast dest: nat_less_not_sym)*

Linearity (totality).

lemma *nat_less_linear:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *m* < *n* ∨ *m* = *n* ∨ *n* < *m*"
unfolding *less_def* **using** *nat_leq_linear* **by** *blast*

lemma *nat_leq_less_linear:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *m* ≤ *n* ∨ *n* < *m*"
using *nat_less_linear* **by** *(auto simp add: less_def)*

lemma *nat_less_cases [case_names less equal greater]:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *(m* < *n* ⟹ *P)* ⟹ *(m* = *n* ⟹ *P)* ⟹ *(n* < *m* ⟹ *P)* ⟹ *P*"
using *nat_less_linear* **by** *blast*

lemma *nat_leq_less_cases:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*; *m* ≤ *n* ⟹ *P*; *n* < *m* ⟹ *P*⟧ ⟹ *P*"
using *nat_leq_cases less_def* **by** *auto*

lemma *nat_not_less:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *(*¬ *m* < *n) = (n* ≤ *m)*"
using *nat_leq_linear* **by** *(auto simp add: less_def dest: nat_leq_antisym)*

lemma *nat_not_less_iff_gr_or_eq:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *(*¬ *m* < *n) = (m* > *n* ∨ *m* = *n)*"
unfolding *nat_not_less* **by** *(auto simp add: less_def)*

lemma *nat_not_less_eq:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *(*¬ *m* < *n) = (n* < Succ*[m])*"
unfolding *nat_not_less* **by** *simp*

lemma *nat_not_leq:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *(*¬ *m* ≤ *n) = (n* < *m)*"
by *(simp add: sym[OF nat_not_less])*

— often useful, but not active by default
lemmas *nat_not_order_simps = nat_not_less nat_not_leq*

lemma *nat_not_leq_eq:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *(*¬ *m* ≤ *n) = (*Succ*[n]* ≤ *m)*"
unfolding *nat_not_leq* **by** *simp*

lemma *nat_neq_iff:*
"⟦*m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ *m* ≠ *n = (m* < *n* ∨ *n* < *m)*"
using *nat_less_linear* **by** *auto*

lemma *nat_neq_lessE:*
"⟦*m* ≠ *n*; *m* ∈ *Nat*; *n* ∈ *Nat*; *(m* < *n* ⟹ *R)*; *(n* < *m* ⟹ *R)*⟧ ⟹ *R*"
by *(auto simp add: nat_neq_iff)*

lemma *nat_antisym_conv1:*

"⟦¬ m < n; m ∈ Nat; n ∈ Nat⟧ ⟹ (m ≤ n) = (m = n)"
by *(auto simp add: nat_not_less elim: nat_leq_antisym)*

lemma *nat_antisym_conv2:*
"⟦m ≤ n; m ∈ Nat; n ∈ Nat⟧ ⟹ (¬ m < n) = (m = n)"
by *(auto simp add: nat_antisym_conv1)*

lemma *nat_antisym_conv3:*
"⟦¬ n < m; m ∈ Nat; n ∈ Nat⟧ ⟹ (¬ m < n) = (m = n)"
by *(auto simp add: nat_not_less elim: nat_leq_antisym)*

lemma *nat_not_lessD:*
"⟦¬ m < n; m ∈ Nat; n ∈ Nat⟧ ⟹ n ≤ m"
by *(simp add: nat_not_less)*

lemma *nat_not_lessI:*
"⟦n ≤ m; m ∈ Nat; n ∈ Nat⟧ ⟹ ¬ m < n"
by *(simp add: nat_not_less)*

lemma *nat_gt0_not0 :*
"n ∈ Nat ⟹ (0 < n) = (n ≠ 0)"
by *(auto simp add: nat_neq_iff)*

lemmas *nat_neq0_conv = sym[OF nat_gt0_not0, standard]*

Introduction properties

lemma *nat_less_Succ_self :*
"n ∈ Nat ⟹ n < Succ[n]" **by** *simp*

lemma *nat_zero_less_Succ :*
"n ∈ Nat ⟹ 0 < Succ[n]" **by** *simp*

Elimination properties.

lemma *nat_less_Succ:*
"⟦m ∈ Nat; n ∈ Nat⟧ ⟹ (m < Succ[n]) = (m < n ∨ m = n)"
by *(simp add: nat_leq_less)*

lemma *nat_less_SuccE:*
assumes *"m < Succ[n]"* **and** *"m ∈ Nat"* **and** *"n ∈ Nat"*
and *"m < n ⟹ P"* **and** *"m = n ⟹ P"*
shows *P*
using *prems* **by** *(auto simp add: nat_leq_less)*

lemma *nat_less_one :*
"n ∈ Nat ⟹ (n < 1) = (n = 0)" **by** *simp*

"Less thanïs antisymmetric, sort of.

lemma *nat_less_antisym:*

```
  "⟦m ∈ Nat; n ∈ Nat; ¬(n < m); n < Succ[m] ⟧ ⟹ m = n"
  by (auto simp add: nat_not_order_simps dest: nat_leq_antisym)
```

Inductive (?) properties.

```
lemma nat_Succ_lessI:
  "⟦m ∈ Nat; n ∈ Nat; m < n; Succ[m] ≠ n⟧ ⟹ Succ[m] < n"
  by (simp add: leq_neq_iff_less)

lemma nat_lessE:
  assumes major: "i < k" and i: "i ∈ Nat" and k: "k ∈ Nat"
  obtains j where "j ∈ Nat" and "i ≤ j" and "k = Succ[j]"
proof -
  from k major have "∃j∈Nat : i ≤ j ∧ k = Succ[j]"
  proof (induct k)
    case 0 with i show ?case by simp
  next
    fix n
    assume n: "n ∈ Nat" and 1: "i < Succ[n]"
      and ih: "i < n ⟹ ∃j∈Nat : i ≤ j ∧ n = Succ[j]"
    from i n 1 have "i < n ∨ i = n" by (simp add: nat_leq_less)
    thus "∃j∈Nat : i ≤ j ∧ Succ[n] = Succ[j]"
    proof
      assume "i < n"
      then obtain j where "j ∈ Nat" and "i ≤ j" and "n = Succ[j]"
        by (blast dest: ih)
      with i have "Succ[j] ∈ Nat" and "i ≤ Succ[j]" and "Succ[n] = Succ[Succ[j]]"
        by auto
      thus ?thesis by blast
    next
      assume "i = n"
      with i show ?thesis by blast
    qed
  qed
  with that show ?thesis by blast
qed

lemma nat_Succ_lessE:
  assumes major: "Succ[i] < k" and i: "i ∈ Nat" and k: "k ∈ Nat"
  obtains j where "j ∈ Nat" and "i < j" and "k = Succ[j]"
  using prems by (auto elim: nat_lessE)

lemma nat_gt0_implies_Succ:
  "⟦0 < n; n ∈ Nat⟧ ⟹ ∃m ∈ Nat: n = Succ[m]"
  by (cases set: Nat, auto)

lemma nat_gt0_iff_Succ:
  "n ∈ Nat ⟹ (0 < n) = (∃m ∈ Nat: n = Succ[m])"
  by (auto dest: nat_gt0_implies_Succ)
```

lemma *nat_less_Succ_eq_0_disj:*
"⟦m ∈ Nat; n ∈ Nat⟧ ⟹ (m < Succ[n]) = (m = 0 ∨ (∃j∈Nat: m = Succ[j] ∧ j < n))"
by *(induct set: Nat, auto)*

lemma *nat_less_antisym_false: "⟦m < n; m ∈ Nat; n ∈ Nat⟧ ⟹ n < m = FALSE"*
unfolding *less_def* **using** *nat_leq_antisym* **by** *auto*

lemma *nat_less_antisym_leq_false: "⟦m < n; m ∈ Nat; n ∈ Nat⟧ ⟹ n ≤ m = FALSE"*
unfolding *less_def* **using** *nat_leq_antisym[of m n]* **by** *auto*

end

A.3. Arithmetic (except division) over natural numbers

theory *NatArith*
imports *NatOrderings*
begin

Additional arithmetic theorems

lemma *Succ_pred [simp]:*
"n ∈ Nat ⟹ n > 0 ⟹ Succ[n – 1] = n"
by *(simp add: diff_Succ_nat nat_gt0_not0)*

lemma *less_imp_Succ_add:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"m < n ⟹ (∃k ∈ Nat: n = Succ[m + k])"* **(is** *"_ ⟹ ?P(n)")*
using *n* **proof** *(induct)*
case *0* **with** *m* **show** *?case* **by** *simp*
next
fix *n*
assume *n: "n ∈ Nat"* **and** *ih: "m<n ⟹ ?P(n)"* **and** *suc: "m < Succ[n]"*
from *suc m n* **show** *"?P(Succ[n])"*
proof *(rule nat_less_SuccE)*
assume *"m<n"*
then obtain *k* **where** *"k ∈ Nat"* **and** *"n = Succ[m + k]"* **by** *(blast dest: ih)*
with *m n* **have** *"Succ[k] ∈ Nat"* **and** *"Succ[n] = Succ[m + Succ[k]]"* **by** *auto*
thus *?thesis* **..**
next
assume *"m=n"*
with *n* **have** *"Succ[n] = Succ[m + 0]"* **by** *simp*
thus *?thesis* **by** *blast*
qed
qed

Monotonicity of Addition

lemma *nat_add_left_cancel_leq [simp]:*
"⟦*k* ∈ *Nat*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (*k* + *m* ≤ *k* + *n*) = (*m* ≤ *n*)"
by *(induct set: Nat, simp_all)*

lemma *nat_add_left_cancel_less [simp]:*
"⟦*k* ∈ *Nat*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (*k* + *m* < *k* + *n*) = (*m* < *n*)"
by *(induct set: Nat, simp_all)*

lemma *nat_add_right_cancel_less [simp]:*
"⟦*k* ∈ *Nat*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (*m* + *k* < *n* + *k*) = (*m* < *n*)"
by *(induct set: Nat, simp_all)*

lemma *nat_add_right_cancel_leq [simp]:*
"⟦*k* ∈ *Nat*; *m* ∈ *Nat*; *n* ∈ *Nat*⟧ ⟹ (*m* + *k* ≤ *n* + *k*) = (*m* ≤ *n*)"
by *(induct set: Nat, simp_all)*

lemma *add_gr_0 [simp]:*
"⟦*n* ∈ *Nat*; *m* ∈ *Nat*⟧ ⟹ (*m* + *n* > 0) = (*m* > 0 ∨ *n* > 0)"
by *(auto dest: nat_gt0_implies_Succ nat_not_lessD)*

lemma *nat_leq_trans_add_left_false [simp]:*
assumes *m:* "*m* ∈ *Nat*" **and** *n:* "*n* ∈ *Nat*" **and** *p:* "*p* ∈ *Nat*"
shows "⟦*m* + *n* ≤ *p*; *p* ≤ *n*⟧ ⟹ (*m* + *n* < *p*) = *FALSE*"
apply *(induct n p rule: diffInduct)*
using *prems* **by** *simp_all*

(Partially) Ordered Groups

— The two following lemmas are just "one half" of *nat_add_left_cancel_leq* and *nat_add_right_cancel_leq* proved above.
lemma *add_leq_left_mono:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*⟧ ⟹ *a* ≤ *b* ⟹ *c* + *a* ≤ *c* + *b*" **by** *simp*

lemma *add_leq_right_mono:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*⟧ ⟹ *a* ≤ *b* ⟹ *a* + *c* ≤ *b* + *c*" **by** *simp*

non-strict, in both arguments

lemma *add_leq_mono:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*; *d* ∈ *Nat*; *a* ≤ *b*; *c* ≤ *d*⟧ ⟹ *a* + *c* ≤ *b* + *d*"
using *nat_leq_trans[of "a + c" "b + c" "b + d"]* **by** *simp*

— Similar for strict less than.
lemma *add_less_left_mono:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*⟧ ⟹ *a* < *b* ⟹ *c* + *a* < *c* + *b*" **by** *simp*

lemma *add_less_right_mono:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*⟧ ⟹ *a* < *b* ⟹ *a* + *c* < *b* + *c*" **by** *simp*

Strict monotonicity in both arguments

lemma *add_less_mono:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*; *d* ∈ *Nat*; *a* < *b*; *c* < *d*⟧ ⟹ *a* + *c* < *b* + *d*"
using *nat_less_trans[of "a + c" "b + c" "b + d"]* **by** *simp*

lemma *add_less_leq_mono:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*; *d* ∈ *Nat*; *a* < *b*; *c* ≤ *d*⟧ ⟹ *a* + *c* < *b* + *d*"
using *nat_less_leq_trans[of "a + c" "b + c" "b + d"]* **by** *simp*

lemma *add_leq_less_mono:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*; *d* ∈ *Nat*; *a* ≤ *b*; *c* < *d*⟧ ⟹ *a* + *c* < *b* + *d*"
using *nat_leq_less_trans[of "a + c" "b + c" "b + d"]* **by** *simp*

Lifting < monotonicity to ≤ monotonicity.

lemma *less_mono_imp_leq_mono:*
assumes *i: "i ∈ Nat"* **and** *j: "j ∈ Nat"* **and** *f: "∀n ∈ Nat : f(n) ∈ Nat"*
shows "⟦⋀*i j*. *i* ∈ *Nat* ⟹ *j* ∈ *Nat* ⟹ *i* < *j* ⟹ *f(i)* < *f(j)*; *i* ≤ *j* ⟧ ⟹ *f(i)* ≤ *f(j)*"
using *prems less_imp_leq[of "f(i)" "f(j)"] nat_leq_less[OF i j]* **by** *auto*

lemma *leq_add1 [simp]:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*⟧ ⟹ *b* ≤ *b* + *a*"
using *add_leq_left_mono[of 0]* **by** *simp*

lemma *leq_add2 [simp]:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*⟧ ⟹ *b* ≤ *a* + *b*"
using *add_leq_right_mono [of 0]* **by** *simp*

lemma *less_iff_Succ_add:*
"⟦*a* ∈ *Nat*; *b* ∈ *Nat*⟧ ⟹ (*a* < *b*) = (∃*k* ∈ *Nat*: *b* = Succ[*a* + *k*])"
by *(auto intro!: less_imp_Succ_add)*

lemma *trans_leq_add1:*
"⟦*a* ≤ *b*; *a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*⟧ ⟹ *a* ≤ *b* + *c*"
by *(auto elim: nat_leq_trans)*

lemma *trans_leq_add2:*
"⟦*a* ≤ *b*; *a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*⟧ ⟹ *a* ≤ *c* + *b*"
by *(auto elim: nat_leq_trans)*

lemma *trans_less_add1:*
"⟦*a* < *b*; *a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*⟧ ⟹ *a* < *b* + *c*"
by *(auto elim: nat_less_leq_trans)*

lemma *trans_less_add2:*
"⟦*a* < *b*; *a* ∈ *Nat*; *b* ∈ *Nat*; *c* ∈ *Nat*⟧ ⟹ *a* < *c* + *b*"
by *(auto elim: nat_less_leq_trans)*

lemma *add_lessD1:*
"⟦*a + b < c; a ∈ Nat; b ∈ Nat; c ∈ Nat*⟧ ⟹ *a < c*"
by *(auto simp: nat_leq_less_trans[of "a" "a + b"])*

lemma *not_add_less1 [simp]:*
"⟦*a ∈ Nat; b ∈ Nat*⟧ ⟹ *(a + b < a) = FALSE*"
by *(auto dest: add_lessD1)*

lemma *not_add_less2 [simp]:*
"⟦*a ∈ Nat; b ∈ Nat*⟧ ⟹ *(b + a < a) = FALSE*"
by *(simp add: add_commute_nat)*

lemma *add_leqD1:*
"⟦*a + c ≤ b; a ∈ Nat; b ∈ Nat; c ∈ Nat*⟧ ⟹ *a ≤ b*"
by *(intro nat_leq_trans[of "a" "a + c" "b"], simp_all)*

lemma *add_leqD2:*
"⟦*a + c ≤ b; a ∈ Nat; b ∈ Nat; c ∈ Nat*⟧ ⟹ *c ≤ b*"
by *(intro nat_leq_trans[of "c" "a + c" "b"], simp_all)*

lemma *add_leqE:*
"⟦*a + c ≤ b; a ∈ Nat; b ∈ Nat; c ∈ Nat*⟧ ⟹ *(a ≤ b ⟹ c ≤ b ⟹ R) ⟹ R*"
by *(blast dest: add_leqD1 add_leqD2)*

lemma *leq_add_left_false [simp]:*
"⟦*a ∈ Nat; b ∈ Nat; 0 < b*⟧ ⟹ *a + b ≤ a = FALSE*"
using *nat_leq_less[of "a + b" a] add_eq_self_zero_nat* **by** *auto*

lemma *less_self_add_not0_nat [intro]:*
"⟦*a ∈ Nat; b ∈ Nat; 0 < b*⟧ ⟹ *a < a + b*"
using *not0_implies_Suc[of b] nat_gt0_not0* **by** *auto*

Used in int_add_assoc* proofs

lemma *add_less_leq_trans1_false [dest]:*
"⟦*m + n < p; p ≤ n; m ∈ Nat; n ∈ Nat; p ∈ Nat*⟧ ⟹ *FALSE*"
by *(drule nat_less_leq_trans[of "m + n" p n], simp_all)*

lemma *add_less_leq_trans2_false [dest]:*
"⟦*m + n < p; p ≤ m; m ∈ Nat; n ∈ Nat; p ∈ Nat*⟧ ⟹ *FALSE*"
by *(drule nat_less_leq_trans[of "m + n" p m], simp_all)*

lemma *add_leq_less_trans1_false [dest]:*
"⟦*m + n ≤ p; p < n; m ∈ Nat; n ∈ Nat; p ∈ Nat*⟧ ⟹ *FALSE*"
by *(drule nat_leq_less_trans[of "m + n" p n], simp_all)*

lemma *add_leq_less_trans2_false [dest]:*
"⟦*m + n ≤ p; p < m; m ∈ Nat; n ∈ Nat; p ∈ Nat*⟧ ⟹ *FALSE*"
by *(drule nat_leq_less_trans[of "m + n" p m], simp_all)*

lemma *add_left_leq_is_0 [dest]:*
"⟦a + b ≤ a; a ∈ Nat; b ∈ Nat⟧ ⟹ b = 0"
by *(simp only: nat_leq_less[of "a+b" a] addIsNat, auto)*

lemma *add_left_leq_is_0_trans [dest]:*
"⟦a + b ≤ c; c ≤ a; a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ b = 0"
by *(drule nat_leq_trans[of "a + b" c a], auto)*

More results about difference over naturals

lemma *pred_less_self [simp]: "a ∈ Nat ⟹ 0 < a ⟹* Pred*[a] < a"*
by *(drule not0_implies_Suc[of a], auto)*

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

lemma *add_diff_inverse:*
assumes *"¬(m < n)" "m ∈ Nat"* **and** *"n ∈ Nat"*
shows *"n + (m – n) = m"*
using *prems* **by** *(induct m n rule: diffInduct, simp_all)*

lemma *le_add_diff_inverse [simp]:*
"⟦m ∈ Nat; n ∈ Nat⟧ ⟹ n ≤ m ⟹ n + (m – n) = m"
by *(simp add: add_diff_inverse nat_not_less)*

lemma *le_add_diff_inverse2 [simp]:*
"⟦m ∈ Nat; n ∈ Nat⟧ ⟹ n ≤ m ⟹ (m – n) + n = m"
by *(simp add: add_commute_nat)*

lemma *Succ_diff_leq:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"n ≤ m ⟹* Succ*[m] – n =* Succ*[m – n]"*
by *(induct m n rule: diffInduct, simp_all add: prems)*

lemma *Succ_diff_less:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"n < m ⟹* Succ*[m] – n =* Succ*[m – n]"*
by *(induct m n rule: diffInduct, simp_all add: prems)*

lemma *diff_less_Succ:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"m – n <* Succ*[m]"*
by *(induct m n rule: diffInduct, simp_all add: prems)*

lemma *diff_leq_self [simp]:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"m – n ≤ m"*
by *(induct m n rule: diffInduct, simp_all add: prems)*

lemma *leq_iff_add:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"m ≤ n = (∃k ∈ Nat: n = m + k)"* **(is** *"?lhs = ?rhs")*
proof -
have *1: "?lhs ⇒ ?rhs"*
proof
assume *mn: "m ≤ n"*
with *m n* **have** *"n = m + (n – m)"* **by** *simp*
with *m n* **show** *"?rhs"* **by** *blast*
qed
from *prems* **have** *2: "?rhs ⇒ ?lhs"* **by** *auto*
from *1 2 prems* **show** *?thesis* **by** *blast*
qed

lemma *less_imp_diff_less [intro]:*
assumes *j: "j ∈ Nat"* **and** *k: "k ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"j < k ⟹ j – n < k"*
apply *(rule nat_leq_less_trans[OF _ _ diffIsNat[OF j n] j k])*
by *(rule diff_leq_self[OF j n])*

lemma *diff_Succ_less [intro]:*
"⟦0 < n; n ∈ Nat; i ∈ Nat⟧ ⟹ n – Succ[i] < n"
by *(cases set: Nat, auto)*

lemma *pred_diff_to_Succ:*
"⟦x ∈ Nat; n ∈ Nat; Succ[Succ[x]] ≤ n⟧ ⟹ Pred[n – x] = Succ[n – Succ[Succ[x]]]"
by *(simp add: Succ_diff_leq[symmetric] diff_Succ_nat[symmetric, of n x]*
del: Succ_diff_leq diff_Succ_nat)

lemma *Succ_add_diff_self1 [simp]:*
"⟦a ∈ Nat; b ∈ Nat⟧ ⟹ Succ[b + a] – a = Succ[b]"
by *(induct set: Nat, simp_all)*

lemma *Succ_add_diff_self2 [simp]:*
"⟦a ∈ Nat; b ∈ Nat⟧ ⟹ Succ[a + b] – a = Succ[b]"
by *(simp add: add_commute_nat[of a])*

Diff and Add associativity. We always try to group the positive arguments

lemma *diff_add_assoc1 [simp]:*
assumes *"c ≤ b"* **and** *"a ∈ Nat" "b ∈ Nat" "c ∈ Nat"*
shows *"a + (b – c) = (a + b) – c"*
using *prems* **by** *(induct b c rule: diffInduct, simp_all)*

lemma *diff_add_assoc2 [simp]:*
assumes *"b ≤ a"* **and** *"a ∈ Nat" "b ∈ Nat" "c ∈ Nat"*
shows *"(a – b) + c = (a + c) – b"*
using *prems* **by** *(induct a b rule: diffInduct, simp_all)*

lemma *diff_add_assoc3 [simp]:*
assumes *"c ≤ b"* **and** *"a ∈ Nat" "b ∈ Nat" "c ∈ Nat"*
shows *"a – (b – c) = (a + c) – b"*
using *prems* **by** *(induct b c rule: diffInduct, simp_all)*

lemma *diff_add_assoc4 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (a – b) – c = a – (b + c)"
by *(rule diffInduct[of a b], auto)*

lemma *diff_is_0_eq [simp]:*
assumes *"m ∈ Nat" "n ∈ Nat"*
shows *"(m – n = 0) = (m ≤ n)"*
by *(induct m n rule: diffInduct, simp_all add: prems)*

lemma *diff_is_0_eq' :*
"⟦m ≤ n; m ∈ Nat; n ∈ Nat⟧ ⟹ m – n = 0" **by** *simp*

lemma *zero_less_diff [simp]:*
assumes *"m ∈ Nat" "n ∈ Nat"*
shows *"(0 < n – m) = (m < n)"*
by *(induct m n rule: diffInduct, simp_all add: prems)*

lemma *less_imp_add_positive:*
assumes *"i < j"* **and** *i: "i ∈ Nat"* **and** *j: "j ∈ Nat"*
shows *"∃k ∈ Nat: 0 < k ∧ i + k = j"*
apply *(rule_tac x="j – i"* **in** *bExI)*
using *prems less_imp_leq* **by** *(simp_all add: zero_less_diff[OF i j])*

Transform diff to add in an inequality

lemma *leq_diff_right_add_left_iff [simp]:*
assumes *"k ≤ n"* **and** *"m ∈ Nat" "n ∈ Nat" "k ∈ Nat"*
shows *"m ≤ n – k = (m + k ≤ n)"*
using *prems* **by** *(induct n k rule: diffInduct, simp_all)*

lemma *leq_diff_right_add_left [dest]:*
assumes *"m ≤ n – k" "k ≤ n"* **and** *"m ∈ Nat" "n ∈ Nat" "k ∈ Nat"*
shows *"(m + k ≤ n)"*
using *prems* **by** *(induct n k rule: diffInduct, simp_all)*

lemma *leq_diff_left_add_right_iff [simp]:*
assumes *"p ≤ n" "m ∈ Nat" "n ∈ Nat" "p ∈ Nat"*
shows *"n – p ≤ m = (n ≤ m + p)"*
using *prems* **by** *(induct n p rule: diffInduct, auto)*

lemma *leq_diff_left_add_right [dest]:*
assumes *"n – p ≤ m"* **and** *"m ∈ Nat" "n ∈ Nat" "p ∈ Nat"*
shows *"n ≤ m + p"*
using *prems* **by** *(induct n p rule: diffInduct, simp_all)*

lemmas *leq_diff_to_add_iff = leq_diff_right_add_left_iff leq_diff_left_add_right_iff*

Transform diff to add in a strict inequality

lemma *less_diff_right_add_left_iff [simp]:*
assumes *"k ≤ n"* **and** *"m ∈ Nat" "n ∈ Nat" "k ∈ Nat"*
shows *"m < n – k = (m + k < n)"*
using *prems* **by** *(induct n k rule: diffInduct, simp_all)*

lemma *less_diff_right_add_left [dest]:*
assumes *"m < n – k" "k ≤ n"* **and** *"m ∈ Nat" "n ∈ Nat" "k ∈ Nat"*
shows *"(m + k < n)"*
using *prems* **by** *(induct n k rule: diffInduct, simp_all)*

lemma *less_diff_left_add_right_iff [simp]:*
assumes *"p ≤ n" "m ∈ Nat" "n ∈ Nat" "p ∈ Nat"*
shows *"n – p < m = (n < m + p)"*
using *prems* **by** *(induct n p rule: diffInduct, auto)*

lemma *less_diff_left_add_right [dest]:*
assumes *"n – p < m"* **and** *"m ∈ Nat" "n ∈ Nat" "p ∈ Nat"*
shows *"n < m + p"*
using *prems* **by** *(induct n p rule: diffInduct, simp_all)*

lemmas *less_diff_to_add_iff = less_diff_right_add_left_iff less_diff_left_add_right_iff*

lemma *leq_diff_trans [intro]:*
"⟦p ≤ m; m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ p – n ≤ m"
apply *(rule nat_leq_trans[of "p – n"])*
using *diff_leq_self* **by** *simp_all*

lemma *leq_diff_right_false [simp]:*
"⟦0 < n; n ≤ m; m ∈ Nat; n ∈ Nat⟧ ⟹ m ≤ m – n = FALSE"
by *(simp add: leq_diff_right_add_left)*

lemma *leq_diff_right_imp_0:*
assumes *h:"n ≤ n – p" "p ≤ n"* **and** *n: "n ∈ Nat"* **and** *p: "p ∈ Nat"*
shows *"p = 0"*
using *p h n* **by** *(induct, auto)*

lemma *less_imp_diff_not0 [intro]:*
"⟦i < j; i ∈ Nat; j ∈ Nat⟧ ⟹ j – i ≠ 0"
proof -
assume *"i < j"* **and** *nat: "i ∈ Nat" "j ∈ Nat"*
hence *"0 < j – i"* **by** *simp*
with *nat* **show** *?thesis* **using** *nat_gt0_not0[of "j – i"]* **by** *auto*
qed

lemma *less_imp_diff_positive:*
assumes *h: "i < j"* **and** *i: "i ∈ Nat"* **and** *j: "j ∈ Nat"*
shows *"⟦i < j; i ∈ Nat; j ∈ Nat⟧ ⟹ ∃k ∈ Nat: j – i = Succ[k]"*
apply *(rule not0_implies_Suc)*
using *less_imp_diff_not0* **by** *simp_all*

lemma *diff_eq_self_is_0 [dest]:*
assumes *1: "n – m = n" "m ≤ n"* **and** *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"m = 0"*
using *m 1* **apply** *(induct, simp_all)*
using *n* **by** *(induct, auto simp: Succ_diff_leq diff_less_Succ[unfolded less_def])*

lemma *diff_eq_self_is_0' [dest]:*
assumes *1: "n – m = n" "m < n"* **and** *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"m = 0"*
using *prems* **by** *auto*

lemma *self_eq_diff_is_0 [dest]:*
"⟦n = n – m; m ≤ n; m ∈ Nat; n ∈ Nat⟧ ⟹ m = 0"
by *(auto dest: sym)*

lemma *self_eq_diff_is_0' [dest]:*
"⟦n = n – m; m < n; m ∈ Nat; n ∈ Nat⟧ ⟹ m = 0"
by *auto*

lemma *diff_left_eq_add_right [simp]:*
assumes *1: "b ≤ a"* **and** *nat: "a ∈ Nat" "b ∈ Nat" "c ∈ Nat"*
shows *"(a – b = c) = (c + b = a)"*
using *prems* **by** *auto*

lemma *diff_left_eq_add_right_sym [simp]:*
assumes *1: "b ≤ a"* **and** *nat: "a ∈ Nat" "b ∈ Nat" "c ∈ Nat"*
shows *"(c = a – b) = (c + b = a)"*
using *prems* **by** *auto*

lemma *diff_cancel_left_nat [simp]:*
"⟦a ≤ c; b ≤ c; a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (c – a = c – b) = (a = b)"
by *(safe, simp add: trans_leq_add1)*

lemma *diff_cancel_left2_nat [simp]:*
"⟦a < c; b < c; a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (c – a = c – b) = (a = b)"
unfolding *less_def* **by** *(rule diff_cancel_left_nat, auto)*

lemma *diff_cancel_eq_nat [simp]:*
"⟦a ≤ c; a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (c – a + b = c) = (a = b)"
by *(auto simp add: trans_leq_add1)*

lemma *diff_cancel_right_nat [simp]:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *c: "c ∈ Nat"*
shows *"⟦c ≤ a; c ≤ b⟧ ⟹ (a – c = b – c) = (a = b)"*
using *prems* **by** *auto*

lemma *diff_cancel_right2_nat [simp]:*
"⟦c < a; c < b; a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (a – c = b – c) = (a = b)"
unfolding *less_def* **by** *(rule diff_cancel_right_nat, auto)*

Monotonicity of Multiplication

lemma *mult_leq_left_mono:*
assumes *1:"a ≤ b"* **and** *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *c: "c ∈ Nat"*
shows *"c * a ≤ c * b"*
using *c 1 a b* **by** *(induct, simp_all add: add_leq_mono)*

lemma *mult_leq_right_mono:*
assumes *1:"a ≤ b"* **and** *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *c: "c ∈ Nat"*
shows *"a * c ≤ b * c"*
using *c 1 a b* **by** *(induct, simp_all add: add_leq_mono)*

≤ monotonicity, BOTH arguments

lemma *mult_leq_mono:*
assumes *"i ≤ j" "k ≤ l"* **and** *"i ∈ Nat" "j ∈ Nat" "k ∈ Nat" "l ∈ Nat"*
shows *"i * k ≤ j * l"*
using *prems mult_leq_right_mono mult_leq_left_mono*
*nat_leq_trans[of "i * k" "j * k" "j * l"]*
by *simp*

strict, in 1st argument

lemma *mult_less_left_mono:*
assumes *1: "i < j" "0 < k"* **and** *i: "i ∈ Nat"* **and** *j: "j ∈ Nat"* **and** *k: "k ∈ Nat"*
shows *"k * i < k * j"*
using *1 k*
proof *(auto simp: nat_gt0_iff_Succ)*
fix *m*
assume *m: "m ∈ Nat"* **and** *"i < j"*
with *m i j* **show** *"*Succ*[m] * i <* Succ*[m] * j"*
by *(induct, simp_all add: add_less_mono)*
qed

lemma *mult_less_right_mono:*
assumes *1: "i < j" "0 < k"* **and** *i: "i ∈ Nat"* **and** *j: "j ∈ Nat"* **and** *k: "k ∈ Nat"*
shows *"i * k < j * k"*
using *1 k*
proof *(auto simp: nat_gt0_iff_Succ)*
fix *m*

```
  assume m: "m ∈ Nat" and "i < j"
  with m i j show "i * Succ[m] < j * Succ[m]"
    by (induct m, simp_all add: add_less_mono)
qed

lemma nat_0_less_mult_iff [simp]:
  "⟦a ∈ Nat; b ∈ Nat⟧ ⟹ (0 < a * b) = (0 < a ∧ 0 < b)"
  by (induct set: Nat, auto)

lemma mult_less_cancel_left [simp]:
  assumes m: "m ∈ Nat" and n: "n ∈ Nat" and k: "k ∈ Nat"
  shows "(k * m < k * n) = (0 < k ∧ m < n)"
proof (auto intro!: mult_less_left_mono simp: prems)
  assume "k * m < k * n"
  with k m n show "0 < k" by (induct, simp_all)
next
  assume "k * m < k * n"
  with m n k show "m < n"
    unfolding less_def[of "k * m" "k * n"]
    apply (auto simp add: nat_not_leq[symmetric])
    using mult_leq_left_mono nat_leq_antisym[of "k * m" "k * n"]
    by auto
qed

lemma mult_less_cancel_right [simp]:
  "⟦m ∈ Nat; n ∈ Nat; k ∈ Nat⟧ ⟹ (m * k < n * k) = (0 < k ∧ m < n)"
  by (simp add: mult_commute_nat)

lemma mult_less_self_left [dest]:
  assumes less: "n*k < n" and n: "n ∈ Nat" and k: "k ∈ Nat"
  shows "k=0"
  using k less n by (cases, auto)

lemma mult_less_self_right [dest]:
  assumes less: "k*n < n" and n: "n ∈ Nat" and k: "k ∈ Nat"
  shows "k=0"
  using k less n by (cases, auto)

lemma mult_leq_cancel_left [simp]:
  "⟦m ∈ Nat; n ∈ Nat; k ∈ Nat⟧ ⟹ (k * m ≤ k * n) = (k = 0 ∨ m ≤ n)"
  by (auto simp add: nat_leq_less nat_gt0_not0)

lemma mult_leq_cancel_right [simp]:
  "⟦m ∈ Nat; n ∈ Nat; k ∈ Nat⟧ ⟹ (m * k ≤ n * k) = (k = 0 ∨ m ≤ n)"
  by (auto simp add: nat_leq_less nat_gt0_not0)

lemma Suc_mult_less_cancel1:
  "⟦m ∈ Nat; n ∈ Nat; k ∈ Nat⟧ ⟹ (Succ[k] * m < Succ[k] * n) = (m < n)"
```

by *(simp del: mult_Succ_left_nat)*

lemma *Suc_mult_leq_cancel1:*
*"⟦m ∈ Nat; n ∈ Nat; k ∈ Nat⟧ ⟹ (Succ[k] * m ≤ Succ[k] * n) = (m ≤ n)"*
by *(simp del: mult_Succ_left_nat)*

lemma *nat_leq_square: "m ∈ Nat ⟹ m ≤ m * m"*
by *(cases set: Nat, auto)*

lemma *nat_leq_cube: "m ∈ Nat ⟹ m ≤ m * (m * m)"*
by *(cases set: Nat, auto)*

Lemma for *gcd*

lemma *mult_eq_self_implies_10:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"*
shows *"(m * n = m) = (n = 1 ∨ m = 0)"* **(is** *"?lhs = ?rhs")*
proof -
have *1: "?lhs ⇒ ?rhs"*
proof
assume *eq: "?lhs"*
with *m n* **have** *"m * n = m * 1"* **by** *simp*
thus *"?rhs"* **unfolding** *mult_cancel1_nat[OF m n oneIsNat]* **.**
qed
from *m n* **have** *2: "?rhs ⇒ ?lhs"* **by** *auto*
from *1 2* **show** *?thesis* **by** *blast*
qed

lemma *mult_add_less_self_nat [dest]:*
assumes *1: "a * b + c < b"* **and** *c: "c ∈ Nat"*
shows *"⟦0 < b; a ∈ Nat; b ∈ Nat⟧ ⟹ a = 0 ∧ c < b"*
using *c 1* **apply** *(induct, simp_all)*
apply *best*
apply *(subgoal_tac "a = 0")*
apply *(auto dest!: nat_Succ_lessD)*
done

isEven predicate

(Used in the definition of negative based exponentiation.)

definition *isEven* **where**
"n ∈ Nat ⟹ isEven(n) ≡ ∃x ∈ Nat : n = x + x"

lemma *boolify_isEven [simp]:*
"n ∈ Nat ⟹ boolify(isEven(n)) = isEven(n)"
unfolding *isEven_def* **by** *simp*

lemma *isEven_isBool [intro!,simp]:*

```
  "n ∈ Nat ⟹ isBool(isEven(n))"
  unfolding isEven_def by simp

lemma isEven_0 [simp]:
  "isEven(0)"
  by (simp add: isEven_def, force)

lemma nat_is_even_or_not:
  "n ∈ Nat ⟹ isEven(n) ∨ ¬ isEven(n)"
  by (auto simp add: isEven_def)

lemma even_is_evenSS [simp]:
  "⟦n ∈ Nat; isEven(n)⟧ ⟹ isEven(Succ[Succ[n]])"
  apply (induct set: Nat)
  by (simp_all add: isEven_def, auto)

lemma evenSS_is_even [dest]:
  "isEven(Succ[Succ[n]]) ⟹ n ∈ Nat ⟹ isEven(n)"
  apply (simp add: isEven_def)
proof safe
  fix x
  assume nat: "n ∈ Nat" "x ∈ Nat" and 1: "Succ[Succ[n]] = x + x"
  hence "Pred[Pred[Succ[Succ[n]]]] = Pred[Pred[x + x]]" by simp
  with nat have "n = Pred[Pred[x + x]]" by simp
  with nat have "n = Pred[x] + Pred[x]"
    apply simp
    by (rule natCases[of x], simp_all add: Pred_add_right_nat)
  with nat show "∃x ∈ Nat : n = x + x" by force
qed

lemma evenSS_eq_even [simp]:
  "n ∈ Nat ⟹ isEven(Succ[Succ[n]]) = isEven(n)"
  by (simp add: boolEqualIff iff_def, auto)

lemma isEven_add_xx [simp]:
  "n ∈ Nat ⟹ isEven(n + n)"
  by (force simp add: isEven_def)

lemma odd_is_even_false [simp]:
  assumes x: "x ∈ Nat" and y: "y ∈ Nat"
  shows "(Succ[y + y] = x + x) = FALSE"
  apply (rule nat_leq_cases[of x y])
  using prems apply (simp_all add: nat_not_leq nat_leq_less)
proof (auto)
  assume h1: "x < y" and h2: "Succ[y + y] = x + x"
  from x y x y h1 h1 have "x + x < y + y" by (rule add_less_mono)
  with y h2[symmetric] show "FALSE" by simp
next
```

```
  assume h1: "y < x" and h2: "Succ[y + y] = x + x"
  from x y h1 have "Succ[y] + y < x + x"
    using nat_Succ_leq_iff_less
    by (simp add: add_leq_less_mono del: add_Succ_left_nat)
  with y h2[symmetric] show "FALSE" by simp
qed

lemma even_is_odd_false [simp]:
  "⟦x ∈ Nat; y ∈ Nat⟧ ⟹ (x + x = Succ[y + y]) = FALSE"
  by (rule natCases[of x], simp_all)

lemma isEven_add_Sxx_false [simp]:
  "n ∈ Nat ⟹ isEven(Succ[n + n]) = FALSE"
  by (force simp add: isEven_def)

lemma isEven_imp_notSucc [simp]:
  "⟦isEven(n); n ∈ Nat⟧ ⟹ isEven(Succ[n]) = FALSE"
  by (force simp add: isEven_def)

lemma evenS_imp_even_false [dest]:
  "⟦isEven(Succ[n]); n ∈ Nat⟧ ⟹ isEven(n) = FALSE"
  by (force simp add: isEven_def)

lemma evenS_even_false [dest]:
  "⟦isEven(Succ[n]); isEven(n); n ∈ Nat⟧ ⟹ FALSE"
  by (force simp add: isEven_def)

lemma notEven_is_evenS [dest!]:
  assumes h: "¬ isEven(n)" and n: "n ∈ Nat"
  shows "isEven(Succ[n])"
  using n h by (induct set: Nat, auto)

lemma notEvenS_is_even [dest!]:
  "⟦¬ isEven(Succ[n]); n ∈ Nat⟧ ⟹ isEven(n)"
  by (rule natCases[of n], auto)

lemma isEven_cases [case_names even odd]:
  assumes n: "n ∈ Nat"
  and even: "isEven(n) ⟹ P" and odd: "isEven(Succ[n]) ⟹ P"
  shows "P"
  apply (cases "isEven(n)")
  using prems by auto
```

isEven on addition

```
lemma even_plus_even_is_even [simp]:
  "⟦isEven(m); isEven(n); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven(m + n)"
  apply (simp add: isEven_def)
proof -
```

```
 assume h1: "∃x ∈ Nat : m = x + x" and h2: "∃x ∈ Nat : n = x + x"
  and nat: "m ∈ Nat" "n ∈ Nat"
 from nat h1 obtain x where x: "x ∈ Nat" "m = x + x" by force
 from nat h2 obtain y where y: "y ∈ Nat" "n = y + y" by force
 from x y have 1: "m + n = x + y + (x + y)"
  apply (simp add: add_assoc_nat)
  apply (simp add: add_assoc_nat[symmetric])
  by (simp add: add_commute_nat)
 with nat x y show "∃x ∈ Nat : m + n = x + x" by auto
qed

lemma evenS_plus_even_is_evenS [simp]:
 "⟦isEven(Succ[m]); isEven(n); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven(Succ[m + n])"
 using even_plus_even_is_even[of "Succ[m]" n] by simp

lemma evenS_plus_even_is_even_false1 [simp]:
 "⟦isEven(Succ[m]); isEven(n); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven(m + n) = FALSE"
 by (auto dest: evenS_plus_even_is_evenS)

lemma evenS_plus_even_is_even_false2 [simp]:
 "⟦isEven(Succ[m]); isEven(n); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven(n + m) = FALSE"
 by (simp add: add_commute_nat)
```

isEven on multiplication

```
lemma even_imp_mult_even [simp]:
 "⟦isEven(m); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven (m * n)"
 apply (simp add: isEven_def, clarsimp)
 by (auto simp: add_mult_distrib_right_nat)

lemma multEven_imp_either_even [dest]:
 "⟦isEven(m * n); m ∈ Nat; n ∈ Nat⟧ ⟹ isEven(m) ∨ isEven(n)"
 apply (rule natCases[of m], simp_all)
 by (cases "isEven(n)", auto)

end
```

A.4. The Integers as a superset of natural numbers

```
theory Integers
imports NatArith
begin
```

Addition

There are basically four cases in the addition ”a + b”: the combinations of positives and negatives of each argument. When both argumentes are positives we use the definition of addition for naturals. Finally, it all reduces to addition or difference of naturals.

axioms
int_add_pn_def: — positive and negative case
"⟦a ∈ Nat; b ∈ Nat⟧ ⟹ a + -.b ≡ IF a ≤ b THEN -.(b – a) ELSE a – b"
int_add_np_def [simp]: — negative and positive case
"⟦a ∈ Nat; b ∈ Nat⟧ ⟹ -.a + b ≡ b + -.a"
int_add_nn_def [simp]: — positive and positive case
"⟦a ∈ Nat; b ∈ Nat⟧ ⟹ -.a + -.b = -.(a + b)"

lemma *int_add_pn_case1 :*
"⟦a ≤ b; a ∈ Nat; b ∈ Nat⟧ ⟹ a + -.b = -.(b – a)"
by *(simp add: int_add_pn_def)*

lemma *int_add_pn_case2 :*
"⟦b < a; a ∈ Nat; b ∈ Nat⟧ ⟹ a + -.b = a – b"
by *(simp add: int_add_pn_def nat_less_antisym_leq_false)*

lemma *int_add_pn_case2' :*
"⟦b ≤ a; a ∈ Nat; b ∈ Nat⟧ ⟹ a + -.b = a – b"
by *(force simp add: int_add_pn_def diff_is_0_eq')*

lemmas *int_add_pn_cases [simp] = int_add_pn_case1 int_add_pn_case2 int_add_pn_case2'*

Closure

lemma *addIsInt [simp]: "⟦m ∈ Int; n ∈ Int⟧ ⟹ m + n ∈ Int"*
by *(rule intCases2[of m n], auto simp: int_add_pn_def)*

Neutral element

lemma *add_0_right_int [simp]: "n ∈ Int ⟹ n + 0 = n"*
by *(rule intCases, auto)*

lemma *add_0_left_int [simp]: "n ∈ Int ⟹ 0 + n = n"*
by *(rule intCases, auto)*

Additive inverse element

lemma *add_inverse_int [simp]: "n ∈ Int ⟹ n + -.n = 0"*
by *(rule intCases2[of n "-.n"], auto)*

lemma *add_inverse2_int [simp]: "n ∈ Int ⟹ -.n + n = 0"*
by *(rule intCases2[of "-.n" n], auto)*

Commutativity

lemma *add_commute_int:*
"⟦m ∈ Int; n ∈ Int⟧ ⟹ m + n = n + m"
by *(rule intCases2[of m n], auto simp: add_commute_nat)*

Cancellativity

lemma *add_right_cancel_nnp_int [simp]:*
"⟦a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (c + -.a = c + -.b) = (a = b)"

unfolding *int_add_pn_def* **by** *(auto, simp add: nat_not_leq)*

lemma *add_right_cancel_ppn_int [simp]:*
"⟦a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (a + -.c = b + -.c) = (a = b)"
unfolding *int_add_pn_def* **by** *(auto simp: trans_leq_add1, simp add: nat_not_leq)*

lemma *add_left_cancel_ppn_int [simp]:*
"⟦a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ -.c + a = -.c + b ⟹ a = b"
by *(simp add: add_commute_nat)*

lemma *add_left_cancel_pnp_int [simp]:*
"⟦a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ (c + a = c + -.b) = (a = -.b)"
unfolding *int_add_pn_def*
using *add_assoc_nat[symmetric]* **by** *(auto simp add: nat_not_leq[unfolded less_def])*

lemma *add_left_cancel_pnn_int [simp]:*
"⟦a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ a + -.c = -.(b + c) = (a = -.b)"
apply *(auto simp add: int_add_pn_def nat_not_leq)*
by *(simp add: add_commute_nat[of b c] add_assoc_nat[of c b a, symmetric])*

lemma *add_left_cancel_npp_int [simp]:*
"⟦a ∈ Nat; b ∈ Nat; c ∈ Nat⟧ ⟹ c + -.a = b + c = (-.a = b)"
apply *(auto, drule sym)* **by** *(simp add: add_commute_nat[of b c])*

lemma *add_right_cancel_int [simp]:*
"⟦a ∈ Int; b ∈ Int; c ∈ Int⟧ ⟹ a + c = b + c = (a = b)"
by *(rule intCases3[of a b c], auto simp add: add_commute_int[of a c] dest: sym)*

lemma *add_left_cancel_int [simp]:*
"⟦a ∈ Int; b ∈ Int; c ∈ Int⟧ ⟹ c + a = c + b = (a = b)"
by *(simp add: add_commute_int)*

Associativity

lemma *int_add_assoc1:*
"⟦m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ m + (n + -.p) = (m + n) + -.p"
proof *(rule nat_leq_less_cases[of p n], auto)*
assume *"m ∈ Nat" "n ∈ Nat" "p ∈ Nat" "p ≤ n"*
thus *"(m + n) – p = m + n + -.p"*
by *(simp add: trans_leq_add2)*
next
assume *"m ∈ Nat" "n ∈ Nat" "p ∈ Nat" "n < p"*
hence *1: "n ≤ p"* **by** *auto*
show *"m + (n + -.p) = m + n + -.p"*
by *(rule nat_leq_cases[of p "m + n"], auto simp: prems 1 add_ac_nat)*
qed

lemma *int_add_assoc2:*
"⟦m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ m + (n + -.p) = (m + -.p) + n"

```
  apply (rule nat_leq_cases[of p n], simp_all)
   apply (rule nat_leq_cases[of p m], simp_all)
    apply (rule nat_leq_less_cases[of "p – m" n], auto simp: add_ac_nat)
   apply (rule nat_leq_cases[of p m], simp_all)
    apply (rule nat_less_cases[of "p – n" m], auto simp: add_ac_nat)
    apply (rule nat_leq_cases[of "p – m" n], simp_all add: add_ac_nat)
done

lemma int_add_assoc3:
  "⟦m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ m + -.(n + p) = m + -.n + -.p"
  apply(rule nat_leq_cases[of "n + p" m], simp_all)
   apply(rule nat_leq_cases[of n m], simp_all)
    apply(rule nat_less_cases[of p "m – n"], auto simp: add_ac_nat)
   apply(rule nat_leq_cases[of n m], simp_all)
    apply(rule nat_less_cases[of p "m – n"], auto simp: add_ac_nat)
done

lemma int_add_assoc4:
  "⟦m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ (n + p) + -.m = (n + -.m) + p"
  apply(rule nat_leq_less_cases[of m "n + p" ], simp_all)
   apply(rule nat_leq_cases[of m n], simp_all)
    apply(rule nat_less_cases[of "m – n" p], auto simp: add_commute_nat)
   apply(rule nat_leq_less_cases[of m n], auto dest!: less_imp_leq)
    apply(rule nat_leq_cases[of "m – n" p], auto simp: add_commute_nat)
done

lemma int_add_assoc5:
  "⟦m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ -.m + (n + -.p) = n + -.m + -.p"
  apply(simp add: add_commute_int[of "-.m" "n + -.p"])
  apply(rule nat_leq_less_cases[of p n], simp_all)
   apply(rule nat_leq_less_cases[of m "n – p"], simp_all)
    apply(rule nat_leq_less_cases[of m n], auto simp: add_commute_nat)
    apply(rule nat_leq_less_cases[of m n], auto simp: add_commute_nat less_def)
    apply(rule nat_leq_less_cases[of m n], auto simp: add_commute_nat)
     apply(rule nat_less_cases[of p "n – m"], auto simp: add_commute_nat less_def)
done

lemma int_add_assoc6:
  "⟦m ∈ Nat; n ∈ Nat; p ∈ Nat⟧ ⟹ -.m + (p + -.n) = p + -.(m + n)"
  using add_commute_int[of "-.m" "p + -.n"] apply simp
  apply(rule nat_leq_less_cases[of n p], simp_all)
   apply(rule nat_leq_cases[of m "p – n"], auto simp: add_commute_nat)
  apply (simp add: less_imp_leq[of p n])
  apply(rule nat_leq_cases[of "m + n" p], auto simp: add_commute_nat)
done

lemma add_assoc_int:
  assumes m: "m ∈ Int" and n: "n ∈ Int" and p: "p ∈ Int"
```

shows *"m + (n + p) = (m + n) + p"*
apply*(rule intCases3[OF m n p], simp_all)*
apply*(rule add_assoc_nat, assumption+)*
apply*(rule int_add_assoc1, assumption+)*
apply*(rule int_add_assoc2, assumption+)*
apply*(rule int_add_assoc3, assumption+)*
apply*(rule int_add_assoc4, assumption+)*
apply*(rule int_add_assoc5, assumption+)*
apply*(rule int_add_assoc6, assumption+)*
apply*(rule add_assoc_nat, assumption+)*
done

Minus sign distributes over addition

lemma *minus_distrib_pn_nat [simp]:*
"⟦m ∈ Nat; n ∈ Nat⟧ ⟹ -.(m + -.n) = n + -.m"
by *(rule nat_leq_cases[of n m], simp_all)*

lemma *minus_distrib_pn_int [simp]:*
"⟦m ∈ Int; n ∈ Int⟧ ⟹ -.(m + -.n) = n + -.m"
by *(rule intCases2[of m n], simp_all add: add_commute_int)*

definition axioms, Int version

lemma *int_add_nn : "⟦a ∈ Int; b ∈ Int⟧ ⟹ -.a + -.b = -.(a + b)"*
by *(rule intCases2[of a b], simp_all)*

Multiplication

axioms
*int_mult_pn_def [simp]: "⟦a ∈ Nat; b ∈ Nat⟧ ⟹ a * -.b = -.(a * b)"*
*int_mult_np_def [simp]: "⟦a ∈ Nat; b ∈ Nat⟧ ⟹ -.a * b = -.(a * b)"*
*int_mult_nn_def [simp]: "⟦a ∈ Nat; b ∈ Nat⟧ ⟹ -.a * -.b = a * b"*

theorems *int_mult_def = int_mult_pn_def int_mult_np_def*

Closure

lemma *multIsInt [simp]: "⟦a ∈ Int; b ∈ Int⟧ ⟹ a * b ∈ Int"*
by *(rule intCases2[of a b], simp_all)*

Neutral element

lemma *mult_0_right_int [simp]: "a ∈ Int ⟹ a * 0 = 0"*
by *(rule intCases[of a], simp_all)*

lemma *mult_0_left_int [simp]: "a ∈ Int ⟹ 0 * a = 0"*
by *(rule intCases[of a], simp_all)*

Commutativity

lemma *mult_commute_int: "⟦a ∈ Int; b ∈ Int⟧ ⟹ a * b = b * a"*
by *(rule intCases2[of a b], simp_all add: mult_commute_nat)*

Identity element

lemma *mult_1_right_int [simp]: "a ∈ Int ⟹ a * 1 = a"*
by *(rule intCases[of a], simp_all)*

lemma *mult_1_left_int [simp]: "a ∈ Int ⟹ 1 * a = a"*
by *(rule intCases[of a], simp_all)*

Associativity

lemma *mult_assoc_int:*
assumes *m: "m ∈ Int"* **and** *n: "n ∈ Int"* **and** *p: "p ∈ Int"*
shows *"m * (n * p) = (m * n) * p"*
by *(rule intCases3[OF m n p], simp_all add: mult_assoc_nat)*

Distributivity

lemma *ppn_distrib_left_nat:* — ppn stands for m=positive, n=positive, p=negative
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *p: "p ∈ Nat"*
shows *"m * (n + -.p) = m * n + -.(m * p)"*
apply *(rule nat_leq_cases[OF p n])*
apply *(rule nat_leq_cases[of "m * p" "m * n"])*
using *prems* **apply** *(simp_all add: diff_mult_distrib2_nat)*
done

lemma *npn_distrib_left_nat:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *p: "p ∈ Nat"*
shows *"-.m * (n + -.p) = -.(m * n) + m * p"*
using *prems* **apply** *(simp add: add_commute_int[of "-.(m * n)" "m * p"])*
apply*(rule nat_leq_cases[OF p n])*
apply*(rule nat_leq_cases[of "m * p" "m * n"], simp_all)*
apply*(simp_all add: diff_mult_distrib2_nat)*
apply*(auto dest: nat_leq_antisym)*
done

lemma *nnp_distrib_left_nat:*
assumes *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *p: "p ∈ Nat"*
shows *"-.m * (-.n + p) = m * n + -.(m * p)"*
using *prems* **apply** *(simp add: add_commute_int[of "-.n" p])*
apply*(rule nat_leq_cases[OF p n])*
apply*(rule nat_leq_cases[of "m * p" "m * n"], simp_all)*
apply*(simp_all add: diff_mult_distrib2_nat)*
apply*(auto dest: nat_leq_antisym)*
done

lemma *distrib_left_int:*
assumes *m: "m ∈ Int"* **and** *n: "n ∈ Int"* **and** *p: "p ∈ Int"*
shows *"m * (n + p) = (m * n + m * p)"*
apply*(rule intCases3[OF m n p],*
simp_all only: int_mult_def int_add_nn_def int_mult_nn_def addIsNat)
apply*(rule add_mult_distrib_left_nat, assumption+)*

```
    apply(rule ppn_distrib_left_nat, assumption+)
    apply(simp add: add_commute_int, rule ppn_distrib_left_nat, assumption+)
    apply(simp only: int_add_nn_def multIsNat add_mult_distrib_left_nat)+
    apply(rule npn_distrib_left_nat, assumption+)
    apply(rule nnp_distrib_left_nat, assumption+)
    apply(simp only: add_mult_distrib_left_nat)
done

lemma pnp_distrib_right_nat:
  assumes m: "m ∈ Nat" and n: "n ∈ Nat" and p: "p ∈ Nat"
  shows "(m + -.n) * p = m * p + -.(n * p)"
apply(rule nat_leq_cases[OF n m])
  apply(rule nat_leq_cases[of "n * p" "m * p"])
  using prems apply(simp_all add: diff_mult_distrib_nat)
done

lemma pnn_distrib_right_nat:
  assumes m: "m ∈ Nat" and n: "n ∈ Nat" and p: "p ∈ Nat"
  shows "(m + -.n) * -.p = -.(m * p) + n * p"
using prems apply (simp add: add_commute_int[of "-.(m * p)" "n * p"])
apply(rule nat_leq_cases[OF n m])
  apply(rule nat_leq_cases[of "n * p" "m * p"])
  using prems apply(simp_all add: diff_mult_distrib_nat)
  apply(auto dest: nat_leq_antisym)
done

lemma npn_distrib_right_nat:
  assumes m: "m ∈ Nat" and n: "n ∈ Nat" and p: "p ∈ Nat"
  shows "(-.m + n) * -.p = m * p + -.(n * p)"
using prems apply (simp add: add_commute_int[of "-.m" n])
apply(rule nat_leq_cases[OF n m])
  apply(rule nat_leq_cases[of "n * p" "m * p"])
  apply(simp_all add: diff_mult_distrib_nat )
  apply(auto dest: nat_leq_antisym)
done

lemma distrib_right_int:
  assumes m: "m ∈ Int" and n: "n ∈ Int" and p: "p ∈ Int"
  shows "(m + n) * p = (m * p + n * p)"
apply(rule intCases3[OF m n p],
  simp_all only: int_mult_def int_add_nn_def int_mult_nn_def addIsNat)
    apply(rule add_mult_distrib_right_nat, assumption+)
    apply(simp only: int_add_nn_def multIsNat add_mult_distrib_right_nat)
    apply(rule pnp_distrib_right_nat, assumption+)
    apply(rule pnn_distrib_right_nat, assumption+)
    apply(simp add: add_commute_int, rule pnp_distrib_right_nat, assumption+)
    apply(rule npn_distrib_right_nat, assumption+)
    apply(simp only: int_add_nn_def multIsNat add_mult_distrib_right_nat)
```

apply*(simp only: add_mult_distrib_right_nat)*
done

Minus sign distributes over multiplication

lemma *minus_mult_left_int: "⟦m ∈ Nat; n ∈ Nat⟧ ⟹ -.(m * n) = -.m * n"*
by *(rule intCases2[of m n], simp_all)*

lemma *minus_mult_right_int: "⟦m ∈ Nat; n ∈ Nat⟧ ⟹ -.(m * n) = m * -.n"*
by *(rule intCases2[of m n], simp_all)*

More results about integer arithmetic

lemma *int_add_add_neg_self1 [simp]: "⟦a ∈ Int; b ∈ Int⟧ ⟹ b + -.(b + a) = -.a"*
by *(simp only: int_add_nn[symmetric], simp add: add_assoc_int)*

lemma *int_add_add_neg_self2 [simp]: "⟦a ∈ Int; b ∈ Int⟧ ⟹ b + -.(a + b) = -.a"*
by *(simp add: add_commute_int[of a])*

lemma *int_add_add_neg_self3 [simp]: "⟦a ∈ Int; b ∈ Int⟧ ⟹ -.(a + b) + b = -.a"*
by *(simp add: add_commute_int)*

lemma *int_add_add_neg_self4 [simp]: "⟦a ∈ Int; b ∈ Int⟧ ⟹ -.(b + a) + b = -.a"*
by *(simp add: add_commute_int)*

lemma *neg_minus_1 [simp]: "n ∈ Nat ⟹ -.n - 1 = -.(*Succ*[n])"*
by *(simp add: int_diff_def)*

More results about order relations over integers

lemma *leq_pn_is0 [dest]: "⟦a ≤ -.b; a ∈ Nat; b ∈ Nat⟧ ⟹ a = 0 ∧ b = 0"*
by *(rule natCases[of b], simp_all)*

lemma *nat_leq_int_is_nat [intro]: "⟦a ≤ b; a ∈ Nat; b ∈ Int⟧ ⟹ b ∈ Nat"*
by *(rule intCases[of b], auto)*

lemma *int_ge0_is_nat [dest]: "⟦0 ≤ b; b ∈ Int⟧ ⟹ b ∈ Nat"*
by *(rule intCases[of b], auto)*

lemma *leq_imp_leq_one [simp]: "⟦a ≤ b; a ∈ Int; b ∈ Int⟧ ⟹ a - 1 ≤ b"*
by *(auto simp add: int_diff_def dest!: trichotomy)*

lemma *leq_imp_less_one [simp]: "⟦a ≤ b; a ∈ Int; b ∈ Int⟧ ⟹ a - 1 < b"*
by *(auto simp add: int_diff_def dest!: trichotomy)*

lemma *less_imp_less_one [simp]: "⟦a < b; a ∈ Int; b ∈ Int⟧ ⟹ a - 1 < b"*
by *(auto simp: less_def[of a b])*

lemma *add_neg_mult_is_Nat_is0 [dest!]:* — used many times on Division.thy
"⟦*a + -.(b * c) ∈ Nat; a ∈ Nat; b ∈ Nat; c ∈ Nat; a < c*⟧ ⟹ *b = 0*"
apply *(auto simp add: int_add_pn_def)*
apply *(auto dest: nat_leq_less_trans)*
apply *(auto simp: nat_not_leq dest: nat_less_trans)*
done

lemma *add_neg_mult_eq0_is0 [dest!]:*
"⟦*a + -.(b * c) = 0; a ∈ Nat; b ∈ Nat; c ∈ Nat; a < c*⟧ ⟹ *a = 0 ∧ b = 0*"
using *add_neg_mult_is_Nat_is0[of a b c]* **by** *simp*

end

A.5. The division operators div and mod on Integers

theory *Division*
imports *Integers Tuples*
begin

consts
div :: "[c,c] ⇒ c" (**infixl** *"\div" 70)*
mod :: "[c,c] ⇒ c" (**infixl** *"%" 70)*

syntax *(xsymbols)*
div :: "[c,c] ⇒ c" (**infixl** *"÷" 70)*

We define *op ÷* and *op %* on *Nat* by means of a characteristic relation with two input arguments *m*, *n* and two output arguments *q*(uotient) and *r*(emainder).

From [TLA+ book, section 18.4]: "Integer division (÷) and modulus (*%*) are defined so that the following two conditions hold, for any integer *a* and positive integer *b*: *a % b ∈ 0 .. (b - 1)* and *a = b * (a ÷ b) + a % b*"

The Division Algorithm

definition *divmod_rel* **where**
"*divmod_rel(a,b,q,r) ≡ a = q * b + r*
∧ 0 < b ∧ 0 ≤ r ∧ r < b"

Existence of *q* and *r* in *divmod_rel*. Case 1 is *a ∈ Nat ∧ q ∈ Nat*. Case 2 is *a ∈ {-.n : n ∈ Nat} ∧ q ∈ {-.n : n ∈ Nat}*.

lemma *divmod_rel_ex_case1:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"*
obtains *q r* **where** *"q ∈ Nat" "r ∈ Nat" "divmod_rel(a,b,q,r)"*
proof -
have *"∃q,r ∈ Nat : a = q * b + r ∧ r < b"*
using *a* **proof** *(induct)*
case *0*
from *b pos* **have** *"0 = 0 * b + 0 ∧ 0 < b"* **by** *simp*

```
    then show ?case by blast
  next
    fix a'
    assume a': "a' ∈ Nat" and ih: "∃q, r ∈ Nat : a' = q*b + r ∧ r < b"
    from ih obtain q' r'
      where h1: "a' = q' * b + r"' and h2: "r' < b"
      and q': "q' ∈ Nat" and r': "r' ∈ Nat" by blast
    show "∃q, r ∈ Nat : Succ[a'] = q * b + r ∧ r < b"
    proof (cases "Succ[r'] < b")
      case True
      from h1 h2 a' q' b r' have "Succ[a'] = q' * b + Succ[r']" by simp
      with True q' r' show ?thesis by blast
    next
      case False
      with b r' have "b ≤ Succ[r']" by (simp add: nat_not_less)
      with r' b h2 have "b = Succ[r']" by (intro nat_leq_antisym, simp+)
      with h1 a' q' r' have "Succ[a'] = Succ[q'] * b + 0" by (simp add: add_ac_nat)
      with pos q' show ?thesis by blast
    qed
  qed
  with pos that show ?thesis by (auto simp add: divmod_rel_def)
qed

lemma divmod_rel_ex_case2:
  assumes a: "a ∈ Nat" and b: "b ∈ Nat" and pos: "0 < b"
  obtains q r where "q ∈ Nat" "r ∈ Nat" "divmod_rel(-.a,b,-.q,r)"
proof -
  have "∃q,r ∈ Nat : -.a = -.q * b + r ∧ r < b"
    using a proof (induct)
    case 0
    from b pos have "-.0 = -.0 * b + 0 ∧ 0 < b" by simp
    then show ?case by blast
  next
    fix a'
    assume a': "a' ∈ Nat" and ih: "∃q, r ∈ Nat : -.a' = -.q*b + r ∧ r < b"
    from ih obtain q' r'
      where h1: "-.a' = -.q' * b + r"' and h2: "r' < b"
      and q': "q' ∈ Nat" and r': "r' ∈ Nat" by blast
    show "∃q, r ∈ Nat : -.(Succ[a']) = -.q * b + r ∧ r < b"
    proof (cases "0 < r"')
      case True
      from a' q' b r' have "-.q' * b + (r' - 1) = -.q' * b + r' - 1"
        apply (simp add: int_diff_def) by (rule add_diff_assoc2_int, simp_all)
      with h1 a' q' b r' have "-.(Succ[a']) = -.q' * b + (r' - 1)"
        by (auto dest: sym)
      with True q' r' h2 b show ?thesis by (auto simp: int_diff_def)
    next
      case False
```

```
    with r' have 1: "r' = 0" by (simp add: nat_gt0_not0)
    from q' b have "-.(Succ[q']) * b = -.(q' * b + b)" by simp
    with h1 a' q' 1 b have "-.(Succ[a']) = -.(Succ[q']) * b + (b - 1)"
     apply (simp add: int_diff_def add_commute_int[of "q' * b" "b"]
      add_commute_int[of "-.(b + q' * b)" "b + -.1"])
     by (simp add: add_assoc_int[symmetric] add_commute_nat)
    with b pos q' show ?thesis
     apply(rule_tac x="Succ[q']" in bExI, simp)
     apply(rule_tac x="b - 1" in bExI, simp add: int_diff_def)
     by auto
   qed
  qed
  with pos that show ?thesis by (auto simp add: divmod_rel_def)
qed
```

divmod_rel has unique solutions.

```
lemma divmod_rel_unique_div_case1:
  assumes 1: "divmod_rel(a,b,q,r)" and 2: "divmod_rel(a,b,q',r')"
    and a: "a ∈ Nat" and b: "b ∈ Nat"
    and q: "q ∈ Nat" and r: "r ∈ Nat" and q': "q' ∈ Nat" and r': "r' ∈ Nat"
  shows "q = q'"
proof -
  from b 1 have pos: "0 < b" and aqr: "a = q*b+r" and rb: "r < b"
    by (auto simp add: divmod_rel_def)
  from b 2 have aqr': "a = q'*b+r'" and rb': "r' < b"
    by (auto simp add: divmod_rel_def)
  {
    fix x y x' y'
    assume nat: "x ∈ Nat" "y ∈ Nat" "x' ∈ Nat" "y' ∈ Nat"
      and eq: "x*b + y = x'*b + y'" and less: "y' < b"
    have "x ≤ x'"
    proof (rule contradiction)
      assume "¬(x ≤ x')"
      with nat have "x' < x" by (simp add: nat_not_leq)
      with nat obtain k where k: "k ∈ Nat" "x = Succ[x'+k]"
        by (auto simp add: less_iff_Succ_add)
      with eq nat b have "x'*b + (k*b + b + y) = x'*b + y'"
        by (simp add: add_mult_distrib_right_nat add_assoc_nat)
      with nat k b have "k*b + b + y = y'" by simp
      with less k b nat have "(k*b + y) + b < b" by (simp add: add_ac_nat)
      with k b nat show "FALSE" by simp
    qed
  }
  from this[OF q r q' r'] this[OF q' r' q r] q q' aqr aqr' rb rb'
  show ?thesis by (intro nat_leq_antisym, simp+)
qed

lemma divmod_rel_unique_div_case2:
```

```
 assumes 1: "divmod_rel(-.a,b,-.q,r)" and 2: "divmod_rel(-.a,b,-.q',r')"
  and a: "a ∈ Nat" and b: "b ∈ Nat"
  and q: "q ∈ Nat" and r: "r ∈ Nat" and q': "q' ∈ Nat" and r': "r' ∈ Nat"
 shows "q = q"'
proof -
 from b 1 have pos: "0 < b" and aqr: "-.a = -.q * b + r" and rb: "r < b"
  by (auto simp add: divmod_rel_def)
 from b 2 have aqr': "-.a = -.q' * b + r"' and rb': "r' < b"
  by (auto simp add: divmod_rel_def)
 {
  fix x y x' y'
  assume nat: "x ∈ Nat" "y ∈ Nat" "x' ∈ Nat" "y' ∈ Nat"
   and eq: "-.x * b + y = -.x' * b + y"' and less: "y' < b"
  have "-.x ≤ -.x"'
   apply (simp add: nat)
  proof (rule contradiction)
   assume "¬(x' ≤ x)"
   with nat have "x < x"' by (simp add: nat_not_leq)
   with nat obtain k where k: "k ∈ Nat" and k1: "x' = Succ[x + k]"
    by (auto simp add: less_iff_Succ_add)
   with eq nat b have "-.(x * b) + y = (-.(x * b) + -.(k * b + b)) + y"'
    by (auto simp add: add_mult_distrib_right_nat add_assoc_nat)
   with nat b k have "y = y' + -.(k * b + b)"
    apply (simp only: add_assoc_int[symmetric] closure)
    apply (simp only: add_left_cancel_int closure)
    by (simp add: add_assoc_int[symmetric])
   with less k b nat have "y + k * b + b < b"
    by (simp add: add_assoc_int[symmetric])
   with k b nat show "FALSE" by simp
  qed
 }
 from this[OF q r q' r'] this[OF q' r' q r] q q' aqr aqr' rb rb'
 show ?thesis by (intro nat_leq_antisym, simp+)
qed

lemma divmod_rel_unique_mod1:
 assumes "divmod_rel(a,b,q,r)" and "divmod_rel(a,b,q',r')"
  and "a ∈ Nat" "b ∈ Nat" "q ∈ Nat" "r ∈ Nat" "q' ∈ Nat" "r' ∈ Nat"
 shows "r = r"'
proof -
 from assms have "q = q"' by (rule divmod_rel_unique_div_case1)
 with assms show ?thesis by (auto simp add: divmod_rel_def)
qed

lemma divmod_rel_unique_mod2:
 assumes "divmod_rel(-.a,b,-.q,r)" and "divmod_rel(-.a,b,-.q',r')"
  and "a ∈ Nat" "b ∈ Nat" "q ∈ Nat" "r ∈ Nat" "q' ∈ Nat" "r' ∈ Nat"
 shows "r = r"'
```

proof -
from *assms* **have** *"q = q"'* **by** *(rule divmod_rel_unique_div_case2)*
with *assms* **show** *?thesis* **by** *(auto simp add: divmod_rel_def)*
qed

Divisibility instantiation by means of *divmod_rel*

definition *divmod* **where**
"divmod(a,b) ≡ CHOOSE z ∈ Int × Nat: divmod_rel(a,b,z[1],z[2])"

lemma *divmodPairEx:*
assumes *a: "a ∈ Int"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"*
shows *"∃z ∈ Int × Nat : divmod_rel(a,b,z[1],z[2])"*
proof *(rule intCases[OF a])*
assume *a: "a ∈ Nat"*
from *a b pos* **obtain** *q r* **where** *"r ∈ Nat" "q ∈ Nat" "divmod_rel(a,b,q,r)"*
by *(rule divmod_rel_ex_case1)*
thus *?thesis* **by** *(auto intro!: inProdI)*
next
fix *a'*
assume *a': "a' ∈ Nat"* **and** *1: "a = -.a"'*
obtain *q r* **where** *"q ∈ Nat" "r ∈ Nat" "divmod_rel(-.a',b,-.q,r)"*
by *(rule divmod_rel_ex_case2[OF a' b pos])*
thus *?thesis*
using *prems 1* **apply** *auto*
by *(rule_tac x="⟨-.q,r⟩"* **in** *bExI, auto)*
qed

lemma *divmodInIntNat [intro!,simp]:*
assumes *"a ∈ Int" "b ∈ Nat" "0 < b"*
shows *"divmod(a,b) ∈ Int × Nat"*
unfolding *divmod_def* **by** *(rule bChooseI2[OF divmodPairEx[OF prems]])*

lemma *divmodInNatNat:*
assumes *"a ∈ Nat"* **and** *"b ∈ Nat"* **and** *"0 < b"*
shows *"divmod(a,b) ∈ Nat × Nat"*
apply *(insert prems)*
unfolding *divmod_def*
apply *(rule bChooseI2, rule divmodPairEx, simp_all add: divmod_rel_def)*
proof *auto*
fix *x*
assume *"x[1] * b + x[2] ∈ Nat" "b ∈ Nat" "x ∈ Int × Nat"*
"0 < b" "0 ≤ x[2]" "x[2] < b"
from *prems* **have** *1: "x[1] ∈ Int" "x[2] ∈ Nat"* **by** *auto*
with *prems* **show** *"x ∈ Nat × Nat"* **by** *(auto elim!: intCases[of "x[1]"])*
qed

lemma *divmodInNegNat:*

```
 assumes "a ∈ Nat" and "b ∈ Nat" and "0 < b"
 shows "divmod(-.a,b) ∈ {-.n : n ∈ Nat} × Nat"
 apply (insert prems)
 unfolding divmod_def
 apply (rule bChooseI2, rule divmodPairEx, simp_all add: divmod_rel_def)
proof auto
 fix x
 assume "x ∈ Int × Nat" "-.a = x[1] * b + x[2]"
  "0 < b" "0 ≤ x[2]" "x[2] < b"
 from prems have 1: "x[1] ∈ Int" and 2: "x[2] ∈ Nat" by auto
 with prems have 3: "x[1] ∈ {-.n : n ∈ Nat}"
  apply simp
  apply (rule intCases[of "x[1]"], simp_all)
  apply (rule natCases[of "a"], simp_all)
  by (drule sym, auto)
 from prems 2 3 show "x ∈ {-.n : n ∈ Nat} × Nat" by auto
qed

lemma divmod_divmod_rel:
 assumes "m ∈ Int" "n ∈ Nat" "0 < n"
 shows "z = divmod(m,n) ⇒ divmod_rel(m,n,z[1],z[2])"
 unfolding divmod_def by (rule bChooseI2[OF divmodPairEx[OF prems]], auto)

lemma divmod_unique_case1:
 assumes h: "divmod_rel(a,b,q,r)"
    and a: "a ∈ Nat" and b: "b ∈ Nat" and pos: "0 < b"
    and q: "q ∈ Nat" and r: "r ∈ Nat"
 shows "divmod(a,b) = ⟨q,r⟩"
 unfolding divmod_def
 apply (rule bChooseI2, rule divmodPairEx)
 using prems
proof (auto elim!: inProdE)
 fix q' r'
 assume h2: "divmod_rel(a, b, q', r')" and q': "q' ∈ Int" and r': "r' ∈ Nat"
 with assms show "q = q'"
  apply (auto elim!: divmod_rel_unique_div_case1[of a b q r])
  by (rule intCases[of q'], auto simp: divmod_rel_def)
next
 fix q' r'
 assume h2: "divmod_rel(a, b, q', r')" and q': "q' ∈ Int" and r': "r' ∈ Nat"
 with assms show "r = r'"
  apply (auto elim!: divmod_rel_unique_mod1[of a b q r])
  by (rule intCases[of q'], auto simp add: divmod_rel_def)
qed

lemma divmod_unique_case2:
 assumes "divmod_rel(-.a,b,-.q,r)"
 and "a ∈ Nat" "b ∈ Nat" "0 < b" "q ∈ Nat" "r ∈ Nat"
```

```
 shows "divmod(-.a,b) = ⟨-.q,r⟩"
 unfolding divmod_def
 apply (rule bChooseI2, rule divmodPairEx)
 using prems
proof (auto elim!: inProdE)
 fix q' r'
 assume 1: "divmod_rel(-.a, b, q', r')" "q' ∈ Int" "r' ∈ Nat"
 with assms show "-.q = q"'
  using divmod_rel_unique_div_case2[of a b q r "-.q"' r'] apply auto
  apply (rule intCases[of q'], auto simp: divmod_rel_def)
  using add_neg_mult_is_Nat_is0[of r q b] by auto
next
 fix q' r'
 assume 1: "divmod_rel(-.a, b, q', r')" "q' ∈ Int" "r' ∈ Nat"
 with assms show "r = r"'
  using divmod_rel_unique_mod2[of a b q r "-.q"' r'] apply auto
  by (rule intCases[of q'], auto simp: divmod_rel_def)
qed

lemma divmod_unique_case3:
 assumes "divmod_rel(a,b,-.q,r)"
 and "a ∈ Nat" and b: "b ∈ Nat" and "0 < b" "q ∈ Nat" "r ∈ Nat"
 shows "divmod(a,b) = ⟨-.q,r⟩"
 unfolding divmod_def
 apply (insert prems)
 apply (rule bChooseI2, rule divmodPairEx, auto simp: elim!: inProdE)
 unfolding divmod_rel_def apply auto
 by (auto elim!: intCases)

lemma divmod_unique_case4:
 assumes "divmod_rel(-.a,b,q,r)"
 and "a ∈ Nat" and b: "b ∈ Nat" and "0 < b" "q ∈ Nat" "r ∈ Nat"
 shows "divmod(-.a,b) = ⟨q,r⟩"
 unfolding divmod_def
 apply (insert prems)
 apply (rule bChooseI2, rule divmodPairEx, auto elim!: inProdE)
 unfolding divmod_rel_def
 by (auto elim!: intCases dest!: neg_eq_nat_is0)

lemma divmod_unique:    — general case
 "⟦divmod_rel(a,b,q,r); a ∈ Int; b ∈ Nat; 0 < b; q ∈ Int; r ∈ Nat⟧
 ⟹ divmod(a,b) = ⟨q,r⟩"
 by (rule intCases2[of a q], simp_all add: divmod_unique_case1
  divmod_unique_case2 divmod_unique_case3 divmod_unique_case4)
```

Operators *op* ÷ and *op*%

defs

div_def: "⟦a ∈ Int; b ∈ Nat; 0 < b⟧ ⟹ a ÷ b ≡ divmod(a,b)[1]"
mod_def: "⟦a ∈ Int; b ∈ Nat; 0 < b⟧ ⟹ a % b ≡ divmod(a,b)[2]"

Closure

lemma *divIsInt [intro!,simp]:*
assumes *"a ∈ Int"* **and** *"b ∈ Nat"* **and** *"0 < b"*
shows *"a ÷ b ∈ Int"*
using *divmodInIntNat[OF prems] prems* **by** *(auto simp add: div_def divmod_def)*

lemma *divIsNat [intro!,simp]:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"*
shows *"a ÷ b ∈ Nat"*
using *divmodInNatNat[OF prems] prems* **by** *(auto simp add: div_def)*

lemma *divIsNeg [intro!,simp]:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"*
shows *"-.a ÷ b ∈ {-.n : n ∈ Nat}"*
using *divmodInNegNat[OF prems] prems* **by** *(force simp add: div_def)*

lemma *modIsNat [intro!,simp]:*
assumes *"a ∈ Int"* **and** *"b ∈ Nat"* **and** *"0 < b"*
shows *"a % b ∈ Nat"*
using *divmodInIntNat[OF prems] prems* **by** *(auto simp add: mod_def divmod_def)*

op ÷ and *op %* is the result of *divmod*.

lemma *divmod_div_mod:*
assumes *"a ∈ Int"* **and** *"b ∈ Nat"* **and** *"0 < b"*
shows *"divmod(a,b) = ⟨a ÷ b, a % b⟩"*
unfolding *div_def[OF prems] mod_def[OF prems]*
using *divmodInIntNat[OF prems]* **by** *force*

lemma *divmod_div_mod_nat:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"*
shows *"divmod(a,b) = ⟨a ÷ b, a % b⟩"*
by *(rule divmod_div_mod[OF natInInt[OF a] b pos])*

op ÷ and *op %* hold in *divmod_rel*.

lemma *divmod_rel_div_mod:*
assumes *"a ∈ Int"* **and** *"b ∈ Nat"* **and** *"0 < b"*
shows *"divmod_rel(a, b, a ÷ b, a % b)"*
using *divmod_divmod_rel[OF prems] divmod_div_mod[OF prems]* **by** *force*

lemma *div_unique:*
assumes *"divmod_rel(a,b,q,r)"*
and *a: "a ∈ Int"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"* **and** *"q ∈ Int"* **and** *"r ∈ Nat"*
shows *"a ÷ b = q"*
unfolding *div_def[OF a b pos]* **using** *divmod_unique prems* **by** *simp*

lemma *mod_unique:*
assumes *"divmod_rel(a,b,q,r)"*
and *a: "a ∈ Int"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"* **and** *"q ∈ Int"* **and** *"r ∈ Nat"*
shows *"a % b = r"*
unfolding *mod_def[OF a b pos]* **using** *divmod_unique prems* **by** *simp*

lemma *mod_less_divisor:*
assumes *"a ∈ Int"* **and** *"b ∈ Nat"* **and** *pos: "0 < b"*
shows *"a % b < b"*
using *prems divmod_rel_div_mod* **by** *(simp add: divmod_rel_def)*

lemma *mod_div_int_equality [simp]:*
*"⟦a ∈ Int; b ∈ Nat; 0 < b⟧ ⟹ (a ÷ b) * b + a % b = a"*
using *divmod_rel_div_mod* **by** *(simp add: divmod_rel_def)*

lemma *mod_div_int_equality2 [simp]:*
*"⟦a ∈ Int; b ∈ Nat; 0 < b⟧ ⟹ b * (a ÷ b) + a % b = a"*
using *mult_commute_int[of "a ÷ b", symmetric]* **by** *simp*

lemma *mod_div_int_equality3 [simp]:*
*"⟦a ∈ Int; b ∈ Nat; 0 < b⟧ ⟹ a % b + (a ÷ b) * b = a"*
using *add_commute_int[of "a % b"]* **by** *simp*

lemma *mod_div_int_equality4 [simp]:*
*"⟦a ∈ Int; b ∈ Nat; 0 < b⟧ ⟹ a % b + b * (a ÷ b) = a"*
using *mult_commute_int[of "a ÷ b", symmetric]* **by** *simp*

"Recursive" computation of *div* and *mod* on Nat.

lemma *divmod_base_case1:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *less: "a < b"*
shows *"divmod(a,b) = ⟨0, a⟩"*
proof -
from *prems* **have** *pos: "0 < b"* **by** *(intro nat_leq_less_trans[of 0 a b], simp+)*
let *?dm = "divmod(a,b)"*
from *a b pos* **have** *1: "divmod_rel(a,b,?dm[1],?dm[2])"*
by *(simp add: divmod_divmod_rel)*
from *a* **have** *a2: "a ∈ Int"* **by** *simp*
from *a2 b pos* **have** *"?dm ∈ Int × Nat"* **by** *(rule divmodInIntNat)*
from *a b pos* **have** *2: "?dm ∈ Nat × Nat"* **by** *(rule divmodInNatNat)*
from *1 2 less b* **have** *4:"?dm[1] * b < b"*
apply*(auto simp add: divmod_rel_def)*
apply*(rule intCases[of "divmod(a, b)[1]"], auto)*
by *(auto elim!: add_lessD1)*
have *3: "?dm[1] = 0"*
apply*(insert 4 2 b)* **by** *(rule intCases[of "divmod(a, b)[1]"], auto)*
with *1 2 a b* **have** *"?dm[2] = a"* **by** *(auto simp add: divmod_rel_def)*
with *3 prodProj[OF 2]* **show** *?thesis* **by** *simp*
qed

lemma *divmod_step_case1:*
 assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"* **and** *geq: "b ≤ a"*
 shows *"divmod(a,b) = ⟨Succ[(a – b) ÷ b], (a – b)% b⟩"*
proof -
 from *a* **have** *ai: "a ∈ Int"* **by** *simp*
 from *ai b pos* **have** *1: "divmod_rel(a, b, a ÷ b, a% b)"*
  by *(rule divmod_rel_div_mod)*
 have *2: "a ÷ b ≠ 0"*
 proof
  assume *"a ÷ b = 0"*
  with *1 ai b pos* **have** *"a < b"* **by** *(auto simp add: divmod_rel_def)*
  with *geq a b* **show** *"FALSE"* **by** *(auto simp add: nat_less_leq_not_leq)*
 qed
 with *a b pos* **obtain** *k* **where** *k1: "k ∈ Nat"* **and** *k2: "a ÷ b = Succ[k]"*
  using *not0_implies_Suc[of "a ÷ b"]* **by** *auto*
 with *1 a b pos* **have** *"a = b + k*b + a% b"*
  by *(auto simp add: divmod_rel_def add_commute_nat)*
 moreover
 from *a b k1 pos geq* **have** *"... – b = k*b + a% b"*
  by *(simp add: diff_add_assoc2[symmetric])*
 ultimately
 have *"a – b = k*b + a% b"* **by** *simp*
 with *pos a b 1* **have** *"divmod_rel(a – b, b, k, a% b)"*
  by *(auto simp add: divmod_rel_def)*
 with *k1 a b pos* **have** *"divmod(a – b, b) = ⟨k, a% b⟩"*
  by *(intro divmod_unique, simp+)*
 moreover
 from *a b pos* **have** *"divmod(a – b, b) = ⟨(a–b) ÷ b, (a–b)% b⟩"*
  by *(intro divmod_div_mod, simp+)*
 ultimately
 have *"a ÷ b = Succ[(a–b) ÷ b]"* **and** *"(a% b) = (a–b)% b"*
  using *a b k2* **by** *auto*
 thus *?thesis* **by** *(simp add: divmod_div_mod[OF ai b pos])*
qed

The ”recursion” equations for *div* and *mod* on Nat.

lemma *div_nat_less [simp]:*
 assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *less: "a < b"*
 shows *"(a ÷ b) = 0"*
proof -
 from *prems* **have** *pos: "0 < b"* **by** *(intro nat_leq_less_trans[of "0" "a" "b"], simp+)*
 from *a* **have** *ai: "a ∈ Int"* **by** *simp*
 from *divmod_base_case1[OF a b less] divmod_div_mod[OF ai b pos]*
 show *?thesis* **by** *simp*
qed

lemma *div_nat_geq:*

assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"* **and** *geq: "b ≤ a"*
shows *"(a ÷ b) =* Succ*[(a – b) ÷ b]"*
using *divmod_step_case1[OF prems] divmod_div_mod_nat[OF a b pos]*
by *simp*

lemma *mod_nat_less [simp]:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *less: "a < b"*
shows *"(a% b) = a"*
proof -
from *prems* **have** *pos: "0 < b"* **by** *(intro nat_leq_less_trans[of "0" "a" "b"], simp+)*
from *a* **have** *ai: "a ∈ Int"* **by** *simp*
from *divmod_base_case1[OF a b less] divmod_div_mod_nat[OF a b pos]*
show *?thesis* **by** *simp*
qed

lemma *mod_nat_geq:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"* **and** *geq: "b ≤ a"*
shows *"(a% b) = (a – b)% b"*
using *divmod_step_case1[OF prems] divmod_div_mod_nat[OF a b pos]*
by *simp*

declare *mod_nat_geq[symmetric, simp]*

Facts about *op ÷* and *op %*

op ÷ distributes over *a + b*, where *a% n = 0* or *b% n = 0* (and *a ∈ Nat*, *b ∈ Nat*)

lemma *div_nat_mult_self1 [simp]:*
assumes *q: "q ∈ Nat"* **and** *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *pos: "0 < n"*
shows *"(q + m * n) ÷ n = m + q ÷ n" (***is** *"?P(m)")*
using *m* **proof** *(induct m)*
from *assms* **show** *"?P(0)"* **by** *simp*
next
fix *k*
assume *k: "k ∈ Nat"* **and** *ih: "?P(k)"*
from *n q k* **have** *"n ≤ q + (k*n + n)"* **by** *(simp add: add_assoc_nat)*
with *n q k pos* **have** *"((q + (k*n + n)) ÷ n) =* Succ*[(q + k*n) ÷ n]"*
by *(simp add: div_nat_geq add_assoc_nat)*
with *ih q m n k pos* **show** *"?P(*Succ*[k])"* **by** *simp*
qed

lemma *div_nat_mult_self2 [simp]:*
assumes *"q ∈ Nat"* **and** *"n ∈ Nat"* **and** *"m ∈ Nat"* **and** *"0 < n"*
shows *"(q + n * m) ÷ n = m + q ÷ n"*
using *prems* **by** *(simp add: mult_commute_nat)*

lemma *div_nat_mult_self3 [simp]:*
assumes *"q ∈ Nat"* **and** *"n ∈ Nat"* **and** *"m ∈ Nat"* **and** *"0 < n"*

shows *"(m * n + q) ÷ n = m + q ÷ n"*
using *prems* **by** *(simp add: add_commute_nat)*

lemma *div_nat_mult_self4 [simp]:*
assumes *"q ∈ Nat"* **and** *"n ∈ Nat"* **and** *"m ∈ Nat"* **and** *"0 < n"*
shows *"(n * m + q) ÷ n = m + q ÷ n"*
using *prems* **by** *(simp add: add_commute_nat)*

lemma *div_0: "⟦b ∈ Nat; 0 < b⟧ ⟹ (0 ÷ b) = 0"*
using *prems* **by** *simp*

lemma *mod_0: "⟦b ∈ Nat; 0 < b⟧ ⟹ 0% b = 0"*
using *prems* **by** *simp*

op% distributes over *a + b*, where *a% n = 0* or *b% n = 0* (and *a ∈ Nat*, *b ∈ Nat*)

lemma *mod_nat_mult_self1 [simp]:*
assumes *q: "q ∈ Nat"* **and** *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *pos: "0 < n"*
shows *"(q + m * n)% n = q% n"*
proof -
from *prems* **have** *"m*n + q = q + m*n"*
by *(simp add: add_commute_nat)*
also from *prems* **have** *"... = ((q + m*n) ÷ n) * n + (q + m*n)% n"*
by *(intro sym[OF mod_div_int_equality], simp+)*
also from *prems* **have** *"... = (m + q ÷ n) * n + (q + m*n)% n"*
by *simp*
also from *prems* **have** *"... = m*n + ((q ÷ n) * n + (q + m*n)% n)"*
by *(simp add: add_mult_distrib_right_nat add_assoc_nat)*
finally have *"q = (q ÷ n) * n + (q + m*n)% n"*
using *prems* **by** *simp*
with *q n pos* **have** *"(q ÷ n) * n + (q + m*n)% n = (q ÷ n) * n + q% n"*
by *simp*
with *prems* **show** *?thesis* **by** *(simp del: mod_div_int_equality)*
qed

lemma *mod_nat_mult_self2 [simp]:*
*"⟦q ∈ Nat; m ∈ Nat; n ∈ Nat; 0 < n⟧ ⟹ (q + n * m)% n = q% n"*
using *prems* **by** *(simp add: mult_commute_nat)*

lemma *mod_nat_mult_self3 [simp]:*
*"⟦q ∈ Nat; m ∈ Nat; n ∈ Nat; 0 < n⟧ ⟹ (m * n + q)% n = q% n"*
using *prems* **by** *(simp add: add_commute_nat)*

lemma *mod_nat_mult_self4 [simp]:*
*"⟦q ∈ Nat; m ∈ Nat; n ∈ Nat; 0 < n⟧ ⟹ (n * m + q)% n = q% n"*
using *prems* **by** *(simp add: add_commute_nat)*

lemma *div_nat_mult_self1_is_id [simp]:*
*"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a * b) ÷ b = a"*

using *prems div_nat_mult_self1 [of 0 a b]* **by** *simp*

lemma *div_nat_mult_self2_is_id [simp]:*
*"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (b * a) ÷ b = a"*
using *prems div_nat_mult_self2[of 0 b a]* **by** *simp*

lemma *mod_nat_mult_self1_is_0 [simp]:*
*"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a * b)% b = 0"*
using *prems mod_nat_mult_self1 [of 0 a b]* **by** *simp*

lemma *mod_nat_mult_self2_is_0 [simp]:*
*"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹(b * a)% b = 0"*
using *prems mod_nat_mult_self2 [of 0 a b]* **by** *simp*

lemma *div_nat_by_1 [simp]:*
"a ∈ Nat ⟹ a ÷ 1 = a"
using *prems div_nat_mult_self1_is_id [of a 1]* **by** *simp*

lemma *mod_nat_by_1 [simp]:*
"a ∈ Nat ⟹ a% 1 = 0"
using *prems mod_nat_mult_self1_is_0 [of a 1]* **by** *simp*

lemma *div_nat_self [simp]:*
"⟦b ∈ Nat; 0 < b⟧ ⟹ b ÷ b = 1"
using *prems div_nat_mult_self1_is_id [of 1]* **by** *simp*

lemma *mod_nat_self [simp]:*
"⟦b ∈ Nat; 0 < b⟧ ⟹ b% b = 0"
using *prems mod_nat_mult_self1_is_0 [of 1]* **by** *simp*

lemma *div_add_self1_case1 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a + b) ÷ b = a ÷ b + 1"
using *div_nat_mult_self1[of a 1 b]* **by** *simp*

lemma *div_add_self2_case1 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (b + a) ÷ b = 1 + a ÷ b"
by *(simp add: add_commute_nat)*

lemma *div_Succ_add_self1_case1 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ Succ*[a + b] ÷ b =* Succ*[a] ÷ b + 1"*
using *div_nat_mult_self1[of "*Succ*[a]" 1 b]* **by** *simp*

lemma *div_Succ_add_self2_case1 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ Succ*[b + a] ÷ b =* Succ*[a] ÷ b + 1"*
by *(simp add: add_commute_nat)*

lemma *mod_nat_add_self1 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a + b)% b = a% b"

using *mod_nat_mult_self1[of a 1 b]* **by** *simp*

lemma *mod_nat_add_self2 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (b + a)% b = a% b"
by *(simp add: add_commute_nat)*

lemma *mod_nat_Succ_add_self1 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ Succ[a + b]% b = Succ[a]% b"
using *mod_nat_mult_self1[of "Succ[a]" 1 b]* **by** *simp*

lemma *mod_nat_Succ_add_self2 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ Succ[b + a]% b = Succ[a]% b"
by *(simp add: add_commute_nat)*

op % distributes over $n - r$ (where $n \in Nat$, $r \in Nat$)

lemma *div_diff_self_less_nat [simp]:*
assumes *r: "r ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *pos: "0 < n" "0 < r"* **and** *"r < n"*
shows *"(n – r) ÷ n = 0"*
apply*(insert prems)*
apply*(rule div_nat_less[of "n – r" n])*
unfolding *nat_leq_less*
using *nat_gt0_implies_Succ[of r]* **by** *auto*

lemma *mod_diff_self_less_nat [simp]:*
assumes *r: "r ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *pos: "0 < n" "0 < r"* **and** *"r < n"*
shows *"(n – r)% n = n – r"*
apply*(insert prems)*
apply*(rule mod_nat_less[of "n – r" n])*
unfolding *nat_leq_less*
using *nat_gt0_implies_Succ[of r]* **by** *auto*

lemma *div_diff_self_Succ_less_nat [simp]:*
assumes *r: "r ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *pos: "0 < n"* **and** *"Succ[r] < n"*
shows *"(n – Succ[r]) ÷ n = 0"*
using *prems* **by** *simp*

lemma *mod_diff_self_Succ_less_nat [simp]:*
assumes *r: "r ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *pos: "0 < n"* **and** *"Succ[r] < n"*
shows *"(n – Succ[r])% n = n – Succ[r]"*
using *prems* **by** *simp*

Cases of $a \div b = k$ and $a \% b = k$

lemma *mod0_imp_ex_divmod_nat:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"*
shows *"a% b = 0 ⇒ (∃q ∈ Nat : divmod_rel(a,b,q,0))"*
using *prems* **apply***(simp add: mod_def divmod_def)*
apply *(rule bChooseI2, rule divmodPairEx, simp_all)*

```
  unfolding divmod_rel_def
  apply (safe, simp_all)
proof -
  fix t
  assume h1: "t[1] * b + 0 ∈ Nat" and h2: "t ∈ Int × Nat"
  with b pos have 1: "t[1] ∈ Nat"
    apply (auto elim!: inProdE)
    by (rule intCases, auto)
  with b show "∃q ∈ Nat : t[1] * b + 0 = q * b"
    by (rule_tac x="t[1]" in bExI, simp_all)
qed

lemma mod0_imp_dvd_nat [dest]:
  assumes "a% b = 0" and nat: "a ∈ Nat" "b ∈ Nat" "0 < b"
  shows "∃q ∈ Nat : a = q * b"
  using prems mod0_imp_ex_divmod_nat[OF nat] by (simp add: divmod_rel_def)

lemma mod_not0_imp_ex_divmod_nat:
  assumes "a% b ≠ 0" and a: "a ∈ Nat" and b: "b ∈ Nat" and pos: "0 < b"
  shows "∃q, r ∈ Nat : divmod_rel(a,b,q,r) ∧ 0 < r"
  apply (insert prems)
  unfolding divmod_rel_def
  apply(rule_tac x="a ÷ b" in bExI, simp_all)
  apply(rule_tac x="a% b" in bExI, simp_all)
  using nat_gt0_not0 mod_less_divisor by simp

lemma mod0_div_mult_self [simp]:
  "⟦a% b = 0; a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a ÷ b) * b = a"
  by (auto dest: mod0_imp_dvd_nat)

lemma div_mult_self_leq [simp]:
  "⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a ÷ b) * b ≤ a"
  apply (cases "a% b = 0", auto)
  by (auto dest: mod_not0_imp_ex_divmod_nat simp: divmod_rel_def)

lemma div_mult_self_gt [intro]:
  "⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ a < a ÷ b * b + b"
  apply (cases "a% b = 0", auto)
  apply (drule mod_not0_imp_ex_divmod_nat)
  by (auto simp: divmod_rel_def)

lemma div_to_leq_less_case1:
  "⟦a ÷ b = k; a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ k * b ≤ a ∧ a < Succ[k] * b"
  by auto
```

Simplification of *op* ÷ and *op* % from Negatives to Nat

```
lemma divmod_div_mod_case2:
```

```
  assumes hyp: "divmod_rel(a,b,q,r)"
     and nat: "a ∈ Nat" "b ∈ Nat" "q ∈ Nat" "r ∈ Nat" and pos: "0 < b"
  shows "divmod_rel(-.(Succ[a]), b, -.(Succ[a ÷ b]), (b – (Succ[a]% b))% b)"
  apply (insert prems)
  unfolding divmod_rel_def div_def
  apply(cases "Succ[a]% b = 0", simp_all)
proof (auto)
  assume h1: "Succ[q * b + r]% b = 0" and h2: "r < b"
  from nat have 1: "Succ[q * b + r] = q * b + Succ[r]" by simp
  from nat h2 have h2': "Succ[r] < Succ[b]" by auto
  from nat pos h1 1 have 2: "∃q ∈ Nat : Succ[r] = q * b"
    using mod0_imp_dvd_nat[of "Succ[r]" b] by (simp del: add_Succ_nat)
  with nat pos h2' obtain q'
    where q': "q' ∈ Nat" "Succ[r] = q' * b" by auto
  with nat h2' pos have "q' = 1"
    by (auto elim!: natCases[of b] natCases[of q'], auto)
  with nat 1 q' show "Succ[q * b + r] = q * b + b" by auto
next
  assume h1: "Succ[q * b + r]% b ≠ 0" and h2: "r < b"
  from nat have 1: "Succ[q * b + r] = q * b + Succ[r]" by simp
  from nat h2 have h2': "Succ[r] ≤ b" by auto
  from nat pos h1 1 have "Succ[r]% b ≠ 0" by (simp del: add_Succ_nat)
  with nat pos have "Succ[r] ≠ b"
    by (auto dest: mod0_imp_dvd_nat[of "Succ[r]" b])
  with nat pos h2' have 3: "Succ[r]% b = Succ[r]"
    by (simp add: nat_leq_less[of "Succ[r]" b]
      del: add_Succ_nat diff_add_assoc1 nat_Succ_leq_iff_less)
  show "-.(Succ[q * b + r]) = (b – Succ[q * b + r]% b)% b + -.(q * b + b)"
    apply (insert nat pos h2')
    apply (rule minusInj)
    apply (simp add: 1 3 add_assoc_int[symmetric]
      del: add_Succ_nat nat_Succ_leq_iff_less diff_add_assoc1)
    by (cases "b = Succ[r]", simp_all add: add_commute_nat)
next
  assume h1: "Succ[q * b + r]% b ≠ 0" and h2: "r < b"
  from nat have 1: "Succ[q * b + r] = q * b + Succ[r]" by simp
  from nat h2 have h2': "Succ[r] ≤ b" by auto
  from nat pos h1 1 have "Succ[r]% b ≠ 0" by (simp del: add_Succ_nat)
  with nat pos have "Succ[r] ≠ b"
    by (auto dest: mod0_imp_dvd_nat[of "Succ[r]" b])
  with nat pos h2' have 3: "Succ[r]% b = Succ[r]"
    by (simp add: nat_leq_less[of "Succ[r]" b]
      del: add_Succ_nat diff_add_assoc1 nat_Succ_leq_iff_less)
  from nat pos h1 h2 show "(b – Succ[q * b + r]% b)% b < b"
    apply (simp add: 1 3 del: add_Succ_nat nat_Succ_leq_iff_less)
    by (cases "b = Succ[r]", simp_all)
qed
```

lemma *divmod_rel_div_mod_case2:*
assumes *"divmod_rel(a,b,q,r)" "a ∈ Nat" "b ∈ Nat" "q ∈ Nat" "r ∈ Nat" "0 < b"*
shows *"divmod(-.(Succ[a]), b) = ⟨-.(Succ[a ÷ b]), (b – (Succ[a] % b)) % b⟩"*
by *(rule divmod_div_mod_case2[THEN divmod_unique_case2, OF prems],*
auto simp: prems)

lemma *div_neg_to_nat:*
assumes *"a ∈ Nat" "b ∈ Nat" "0 < b"*
shows *"-.(Succ[a]) ÷ b = -.(Succ[a ÷ b])"*
apply *(rule div_unique)*
apply *(rule divmod_div_mod_case2)*
apply *(rule divmod_rel_div_mod[of a b])*
using *prems* **by** *auto*

lemma *mod_neg_to_nat:*
assumes *"a ∈ Nat" "b ∈ Nat" "0 < b"*
shows *"-.(Succ[a]) % b = (b – (Succ[a] % b)) % b"*
apply *(rule mod_unique)*
apply *(rule divmod_div_mod_case2)*
apply *(rule divmod_rel_div_mod[of a b])*
using *prems* **by** *auto*

Facts about *op ÷* and *op %* on negatives numbers

lemma *div_mult_self1_case2 [simp]:*
assumes *q: "q ∈ Nat"* **and** *m: "m ∈ Nat"* **and** *n: "n ∈ Nat"* **and** *pos: "0 < n"*
shows *"(-.(Succ[q]) + -.m * n) ÷ n = -.m + -.(Succ[q]) ÷ n" (***is** *"?P(m)")*
using *m* **proof** *(induct m)*
from *assms* **show** *"?P(0)"* **by** *simp*
next
fix *k*
assume *k: "k ∈ Nat"* **and** *ih: "?P(k)"*
from *q k n pos* **have** *1: "-.(Succ[q] + Succ[k] * n) ÷ n = -.(Succ[(q + Succ[k] * n) ÷ n])"*
by *(simp add: div_neg_to_nat)*
from *q k n pos* **have** *2: "-.(Succ[k]) + -.(Succ[q]) ÷ n = -.(Succ[k] + Succ[q ÷ n])"*
by *(simp add: div_neg_to_nat)*
from *q k n pos 1 2* **show** *"?P(Succ[k])"*
by *(simp del: add_Succ_left_nat mult_Succ_left_nat)*
qed

lemma *div_mult_self2_case2 [simp]:*
assumes *"q ∈ Nat"* **and** *"n ∈ Nat"* **and** *"m ∈ Nat"* **and** *"0 < n"*
shows *"(-.(Succ[q]) + n * -.m) ÷ n = -.m + -.(Succ[q]) ÷ n"*
using *prems* **by** *(simp add: mult_commute_int[of n "-.m"] del: int_mult_def)*

lemma *mod_mult_self1_case2 [simp]:*
assumes *"q ∈ Nat"* **and** *"m ∈ Nat"* **and** *"n ∈ Nat"* **and** *"0 < n"*
shows *"(-.(Succ[q]) + -.m * n) % n = (n – (Succ[q] % n)) % n"*

using *prems mod_neg_to_nat[of "q + m * n" n]*
by *(simp add: add_Succ_left_nat[symmetric] del: add_Succ_left_nat)*

lemma *mod_mult_self2_case2 [simp]:*
assumes *"q ∈ Nat"* **and** *"m ∈ Nat"* **and** *"n ∈ Nat"* **and** *"0 < n"*
shows *"(-.(Succ[q]) + n * -.m)% n = (n – (Succ[q]% n))% n"*
using *prems* **by** *(simp add: mult_commute_int[of n "-.m"] del: int_mult_def)*

lemma *div_neg1 [simp]:*
"⟦b ∈ Nat; 0 < b⟧ ⟹ -.1 ÷ b = -.1"
apply *(rule div_unique[***where** *r="b – 1"])*
unfolding *divmod_rel_def div_def* **by** *auto*

lemma *mod_neg1 [simp]:*
"⟦b ∈ Nat; 0 < b⟧ ⟹ -.1% b = b – 1"
apply *(rule mod_unique[***where** *q="-.1"])*
unfolding *divmod_rel_def mod_def* **by** *auto*

lemma *div_neg_self [simp]:*
"⟦b ∈ Nat; 0 < b⟧ ⟹ -.b ÷ b = -.1"
apply *(rule div_unique[***where** *r="0"])*
unfolding *divmod_rel_def div_def* **by** *auto*

lemma *mod_neg_self [simp]:*
"⟦b ∈ Nat; 0 < b⟧ ⟹ -.b% b = 0"
apply *(rule mod_unique[***where** *q="-.1"])*
unfolding *divmod_rel_def mod_def* **by** *auto*

lemma *div_mult_neg_self1_is_id [simp]:*
*"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(a * b) ÷ b = -.a"*
apply *(induct set: Nat, auto)*
apply *(rule_tac n="n * b + b"* **in** *natCases, simp_all)*
by *(rule_tac n="n * b"* **in** *natCases, auto simp add: div_neg_to_nat)*

lemma *div_mult_neg_self2_is_id [simp]:*
*"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(b * a) ÷ b = -.a"*
by *(auto simp add: mult_commute_nat)*

lemma *mod_mult_neg_self1_is_0 [simp]:*
*"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(a * b)% b = 0"*
apply *(induct set: Nat, auto)*
apply *(rule_tac n="n * b + b"* **in** *natCases, simp_all add: mod_neg_to_nat)*
by *(rule_tac n="n * b"* **in** *natCases, auto simp add: mod_neg_to_nat)*

lemma *mod_mult_neg_self2_is_0 [simp]:*
*"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(b * a)% b = 0"*
by *(auto simp add: mult_commute_nat)*

```
lemma div_neg_by_1 [simp]:
  "m ∈ Nat ⟹ -.m ÷ 1 = -.m"
  using div_mult_neg_self1_is_id[of m 1] by simp

lemma mod_neg_by_1 [simp]:
  "m ∈ Nat ⟹ -.m % 1 = 0"
  using mod_mult_neg_self1_is_0[of m 1] by simp

lemma div_neg_diff_self1 [simp]:
  assumes "a ∈ Nat" "b ∈ Nat" "0 < b" "a < b"
  shows "-.(b – a) ÷ b = -.1"
  apply (insert prems)
  apply (rule natCases[of a], auto simp add: diff_Succ_nat)
proof -
  fix x
  assume nat: "x ∈ Nat" "b ∈ Nat" and h: "Succ[x] < b" and pos: "0 < b"
  from nat h have 1: "Succ[Succ[x]] ≤ b"
    using nat_less_iff_Succ_leq[symmetric] by simp
  with nat have 2: "Pred[b – x] = Succ[b – Succ[Succ[x]]]"
    using pred_diff_to_Succ by simp
  from nat pos 1 show "-.(Pred[b – x]) ÷ b = -.1"
    using div_neg_to_nat by (auto simp add: 2)
qed

lemma mod_neg_diff_self1 [simp]:
  "⟦a ∈ Nat; b ∈ Nat; 0 < b; a < b⟧ ⟹ -.(b – a) % b = a"
  apply (insert prems)
  apply (rule natCases[of a], auto simp add: diff_Succ_nat)
proof -
  fix x
  assume nat: "x ∈ Nat" "b ∈ Nat" and pos: "0 < b" and h: "Succ[x] < b"
  from nat h have 1: "Succ[Succ[x]] ≤ b"
    using nat_less_iff_Succ_leq[symmetric] by simp
  from nat 1 pos have 2: "Succ[b – Succ[Succ[x]]] < b"
    by (simp add: diff_Succ_nat[of b "Succ[x]"] diff_Succ_less)
  from nat 1 have 3: "Pred[b – x] = Succ[b – Succ[Succ[x]]]"
    by (rule pred_diff_to_Succ)
  from nat pos 1 show "-.(Pred[b – x]) % b = Succ[x]"
    apply (simp add: 2 3 mod_neg_to_nat[of "b – Succ[Succ[x]]" b])
    using nat_gt0_implies_Succ[of b] by auto
qed

lemma div_add_self_case2 [simp]:
  assumes a: "a ∈ Nat" and b: "b ∈ Nat" and pos: "0 < b"
  shows "(-.a + b) ÷ b = -.a ÷ b + 1"
  apply (insert prems)
  apply (simp add: int_add_pn_def)
  apply (cases "b ≤ a", auto simp add: condElse nat_not_leq)
```

apply *(simp add: nat_leq_less, auto)*
apply *(auto simp add: less_iff_Succ_add[of b a] less_iff_Succ_add[of a b])*
by *(auto elim!: natCases[of a] simp: div_neg_to_nat)*

lemma *div_add_self_case3 [simp]:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"*
shows *"(a + -.b) ÷ b = a ÷ b + -.1"*
apply *(insert prems)*
apply *(simp add: int_add_pn_def)*
apply *(cases "a ≤ b", auto simp add: condElse nat_not_leq)*
apply *(simp_all add: nat_leq_less, auto)*
apply *(auto dest!: div_to_leq_less_case1)*
apply *(drule nat_less_antisym_false[of b a], simp+)*
by *(auto simp add: less_iff_Succ_add[of b a])*

lemma *div_add_self1_case4 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(a + b) ÷ b = -.a ÷ b + -.1"
apply *(induct set: Nat, auto)*
using *div_mult_self1_case2[of _ 1 b]* **by** *(simp add: add_commute_int)*

lemma *div_add_self2_case4 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(b + a) ÷ b = -.a ÷ b + -.1"
by *(auto simp add: add_commute_nat)*

lemma *mod_add_self_case2 [simp]:*
assumes *a: "a ∈ Nat"* **and** *b: "b ∈ Nat"* **and** *pos: "0 < b"*
shows *"(-.a + b)% b = -.a% b"*
apply *(insert prems)*
apply *(simp add: int_add_pn_def)*
apply *(cases "b ≤ a", auto simp add: condElse nat_not_leq)*
apply *(simp add: nat_leq_less, auto)*
apply *(auto simp add: less_iff_Succ_add[of b a] less_iff_Succ_add[of a b])*
by *(auto elim!: natCases[of a] simp: mod_neg_to_nat)*

lemma *mod_add_self_case3 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ (a + -.b)% b = a% b"
apply *(simp add: int_add_pn_def)*
apply *(cases "a ≤ b", auto simp add: condElse nat_not_leq)*
by *(auto simp add: nat_leq_less)*

lemma *mod_add_self1_case4 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(a + b)% b = -.a% b"
apply *(induct set: Nat, auto)*
using *mod_mult_self1_case2[of _ 1 b]*
by *(simp add: add_commute_int mod_neg_to_nat)*

lemma *mod_add_self2_case4 [simp]:*
"⟦a ∈ Nat; b ∈ Nat; 0 < b⟧ ⟹ -.(b + a)% b = -.a% b"

by *(auto simp add: add_commute_nat)*

end

Apéndice B

Caso de ejemplo: verificación de las invariantes del Algoritmo de la Panadería

En este apéndice se encuentra la teoría de Isabelle que incluye la especificación del Algoritmo de la Panadería [Lam74] y las pruebas de su invariante de tipo ($Init \wedge \Box[Next]_{vars} \Rightarrow \Box TypeOK$) y su invariante general ($Init \wedge \Box[Next]_{vars} \Rightarrow \Box Inv$). La teoría importada *Constant* contiene a su vez a todas las teorías de Isabelle/TLA$^+$.

B.1. AtomicBakeryG example

theory *AtomicBakeryG*
imports *Constant*
begin

consts
P :: "c"
defaultInitValue :: "c"

axioms
PInNat: "P ⊆ Nat"

abbreviation *ProcSet* **where**
"ProcSet ≡ P"

definition *Init* **where**
"Init(unread,max,flag,pc,num,nxt) ≡
num = [i ∈ P ↦ 0]
∧ flag = [i ∈ P ↦ FALSE]
∧ unread = [self ∈ P ↦ defaultInitValue]
∧ max = [self ∈ P ↦ defaultInitValue]
∧ nxt = [self ∈ P ↦ defaultInitValue]
∧ pc = [self ∈ ProcSet ↦ "p1"]"

definition *p1* **where**
"p1(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡
pc[self] = "p1"
∧ unread' = [unread EXCEPT ![self] = P \ {self}]
∧ max' = [max EXCEPT ![self] = 0]
∧ flag' = [flag EXCEPT ![self] = TRUE]
∧ pc' = [pc EXCEPT ![self] = "p2"]
∧ num' = num ∧ nxt' = nxt"

definition *p2* **where**
"p2(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡
pc[self] = "p2"
∧ (IF unread[self] ≠ {}
THEN ∃i ∈ unread[self] :
(unread' = [unread EXCEPT ![self] = unread[self] \ {i}]
∧ (IF num[i] > max[self]
THEN max' = [max EXCEPT ![self] = num[i]]
ELSE (max' = max)))
∧ pc' = [pc EXCEPT ![self] = "p2"]
ELSE pc' = [pc EXCEPT ![self] = "p3"]
∧ unread' = unread ∧ max' = max)
∧ num' = num ∧ flag' = flag ∧ nxt' = nxt"

definition *p3* **where**
"p3(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡
pc[self] = "p3"
∧ num' = [num EXCEPT ![self] = max[self] + 1]
∧ pc' = [pc EXCEPT ![self] = "p4"]
∧ flag' = flag ∧ unread' = unread ∧ max' = max ∧ nxt' = nxt"

definition *p4* **where**
"p4(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡
pc[self] = "p4"
∧ flag' = [flag EXCEPT ![self] = FALSE]
∧ unread' = [unread EXCEPT ![self] = P \ {self}]
∧ pc' = [pc EXCEPT ![self] = "p5"]
∧ num' = num ∧ max' = max ∧ nxt' = nxt"

definition *p5* **where**
"p5(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡
pc[self] = "p5"
∧ (IF unread[self] ≠ {}
THEN (∃i ∈ unread[self] : nxt' = [nxt EXCEPT ![self] = i])
∧ ¬flag[nxt'[self]]
∧ pc' = [pc EXCEPT ![self] = "p6"]
ELSE pc' = [pc EXCEPT ![self] = "p7"]
∧ nxt' = nxt)
∧ num' = num ∧ flag' = flag ∧ unread' = unread ∧ max' = max"

definition *p6* **where**
"p6(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡
pc[self] = "p6"
∧ (num[nxt[self]] = 0
∨ (IF self > nxt[self]
THEN num[nxt[self]] > num[self]
ELSE num[nxt[self]] ≥ num[self]))
∧ unread' = [unread EXCEPT ![self] = unread[self] \ {nxt[self]}]
∧ pc' = [pc EXCEPT ![self] = "p5"]
∧ num' = num ∧ flag' = flag ∧ max' = max ∧ nxt' = nxt"

definition *p7* **where** — Critical section
"p7(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡
pc[self] = "p7"
∧ TRUE
∧ pc' = [pc EXCEPT ![self] = "p8"]
∧ num' = num ∧ flag' = flag ∧ unread' = unread ∧ max' = max ∧ nxt' = nxt"

definition *p8* **where**
"p8(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡
pc[self] = "p8"
∧ num' = [num EXCEPT ![self] = 0]
∧ pc' = [pc EXCEPT ![self] = "p1"]
∧ flag' = flag ∧ unread' = unread ∧ max' = max ∧ nxt' = nxt"

definition *p* **where**
"p(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡
p1(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')
∨ p2(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')
∨ p3(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')
∨ p4(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')
∨ p5(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')
∨ p6(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')
∨ p7(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')
∨ p8(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"

definition *Next* **where**
"Next(unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt') ≡ (∃ self ∈ P :
p(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt'))
∨ ((∀ self ∈ ProcSet : pc[self] = "Done")
∧ num' = num ∧ flag' = flag ∧ pc' = pc
∧ unread' = unread ∧ max' = max ∧ nxt' = nxt)"

definition *MutualExclusion* **where**
"MutualExclusion(pc) ≡ ∀ i,j ∈ P : (i ≠ j) ⇒ ¬(pc[i] = "p7" ∧ pc[j] = "p7")"

definition *TypeOK* **where**
"TypeOK(unread,max,flag,pc,num,nxt) ≡
num ∈ [P → Nat]
∧ flag ∈ [P → BOOLEAN]
∧ unread ∈ [P → SUBSET P ∪ {defaultInitValue}]
∧ (∀ i ∈ P :
pc[i] ∈ {"p2", "p5", "p6"} ⇒ unread[i] ⊆ P
∧ i ∉ unread[i])
∧ max ∈ [P → Nat ∪ {defaultInitValue}]
∧ (∀ j ∈ P : (pc[j] ∈ {"p2", "p3"}) ⇒ max[j] ∈ Nat)
∧ nxt ∈ [P → P ∪ {defaultInitValue}]
∧ (∀ i ∈ P : (pc[i] = "p6") ⇒ nxt[i] ∈ P \ {i})
∧ pc ∈ [P → {"p1","p2","p3","p4","p5","p6","p7","p8"}]"
— The type invariant in p6 should be *∧ (∀ i ∈ P : (pc[i] = "p6") ⇒ nxt[i] ∈ unread[i] \ {i})* but it works anyway as it is.

definition *GG* **where**
"GG(j,i, num) ≡ IF j > i THEN num[i] > num[j] ELSE num[i] ≥ num[j]"

definition *After* **where**
"After(i,j,unread,max,flag,pc,num,nxt) ≡
num[j] > 0
∧ (pc[i] = "p1"
∨ (pc[i] = "p2"
∧ (j ∈ unread[i]
∨ max[i] ≥ num[j]))
∨ (pc[i] = "p3"
∧ max[i] ≥ num[j])
∨ (pc[i] ∈ {"p4", "p5", "p6"}
∧ GG(j,i, num)
∧ (pc[i] ∈ {"p5", "p6"} ⇒ j ∈ unread[i])))"

definition *IInv* **where**
"IInv(i,unread,max,flag,pc,num,nxt) ≡
((num[i] = 0) = (pc[i] ∈ {"p1", "p2", "p3"}))
∧ (flag[i] = (pc[i] ∈ {"p2", "p3", "p4"}))
∧ (pc[i] ∈ {"p5", "p6"} ⇒
(∀ j ∈ (P \ unread[i]) \ {i} : After(j,i,unread,max,flag,pc,num,nxt)))
∧ (pc[i] = "p6"
∧ ((pc[nxt[i]] = "p2") ∧ i ∉ unread[nxt[i]]
∨ (pc[nxt[i]] = "p3"))
⇒ max[nxt[i]] ≥ num[i])
∧ ((pc[i] ∈ {"p7", "p8"}) ⇒ (∀ j ∈ P \ {i} : After(j,i,unread,max,flag,pc,num,nxt)))"

definition *Inv* **where**
"Inv(unread,max,flag,pc,num,nxt) ≡
TypeOK(unread,max,flag,pc,num,nxt)
∧ (∀ i ∈ P : IInv(i,unread,max,flag,pc,num,nxt))"

lemma *procInNat:*
 assumes *"i ∈ P"* **shows** *"i ∈ Nat"*
using *assms* **by** *(blast dest: subsetD[OF PInNat])*

B.1.1. Type Invariant proof

theorem *InitImpliesTypeOK:*
 "Init(unread,max,flag,pc,num,nxt) ⟹ TypeOK(unread,max,flag,pc,num,nxt)"
 unfolding *Init_def TypeOK_def*
by *(auto intro!: functionInFuncSet)*

theorem *TypeOKInvariant:*
 assumes *type: "TypeOK(unread,max,flag,pc,num,nxt)"* **and** *self: "self ∈ P"*
  and *p: "p(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"*
 shows *"TypeOK(unread',max',flag',pc',num',nxt')"*
using *p* **proof** *(auto simp add: p_def)*
 assume *"p1(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"*
 with *type* **show** *?thesis*
  by *(clarsimp simp: TypeOK_def p1_def, auto)*
next
 assume *p2: "p2(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"*
 show *?thesis*
 proof *(cases "unread[self] = {}")*
  case *True* **with** *type p2* **show** *?thesis*
   by *(clarsimp simp: TypeOK_def p2_def, auto)*
 next
  case *False*
  from *type* **have** *"isAFcn(pc)"*

   by *(force simp add: TypeOK_def)*
  with *False p2* **obtain** *i* **where**
   i: "pc[self] = "p2"" "i ∈ unread[self]"
    "unread' = [unread EXCEPT ![self] = unread[self] \ {i}]"
    "max' = (IF num[i] > max[self] THEN [max EXCEPT ![self] = num[i]] ELSE max)"
    "pc' = pc"
    "num' = num" "flag' = flag" "nxt' = nxt"
   by *(auto simp: p2_def)*
  with *type*
  have *triv: "num' ∈ [P → Nat]" "flag' ∈ [P → BOOLEAN]"*
      "nxt' ∈ [P → P ∪ {defaultInitValue}]"
      "pc' ∈ [P → {"p1","p2","p3","p4","p5","p6","p7","p8"}]"
      "∀ i ∈ P : (pc'[i] = "p6") ⇒ nxt'[i] ∈ P \ {i}"
   by *(auto simp: TypeOK_def)*
  from *i type self* **have** *u: "unread[self] ⊆ ProcSet"*
   by *(auto simp add: TypeOK_def)*
  with *i type* **have** *u': "unread' ∈ [P → SUBSET P ∪ {defaultInitValue}]"*

```
    by (auto simp: TypeOK_def)
  from u i type have u": "∀ i ∈ P : pc'[i] ∈ {"p2", "p5", "p6"} ⇒ unread'[i] ⊆ P ∧ i ∉ unread'[i]"
    by (clarsimp simp: TypeOK_def, blast)
  from u type i have n: "num[i] ∈ Nat"
    by (auto simp: TypeOK_def)
  with i type have m': "max' ∈ [P → Nat ∪ {defaultInitValue}]"
    by (auto simp: TypeOK_def)
  from n i type have m": "∀ j ∈ P : pc'[j] ∈ {"p2","p3"} ⇒ max'[j] ∈ Nat"

    by (cases "max[self] < num[i]", auto simp: TypeOK_def  condElse)
  from triv u' u" m' m" show ?thesis
    by (auto simp add: TypeOK_def)
 qed
next
 assume "p3(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"
 with type show ?thesis
  by (clarsimp simp: TypeOK_def p3_def, auto)
next
 assume "p4(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"
 with type show ?thesis
  by (clarsimp simp: TypeOK_def p4_def, auto)
next
 assume p5: "p5(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"
 show ?thesis
 proof (cases "unread[self] = {}")
  case True with type p5 show ?thesis
   by (clarsimp simp: TypeOK_def p5_def, auto)
 next
  case False
  with p5 obtain i where
   i: "pc[self] = "p5"" "i ∈ unread[self]" "nxt' = [nxt EXCEPT ![self] = i]"
     "¬flag[nxt'[self]]" "pc' = [pc EXCEPT ![self] = "p6"]"
     "num' = num" "flag' = flag" "unread' = unread" "max' = max"
   by (auto simp: p5_def)
  with type show ?thesis
   by (clarsimp simp: TypeOK_def, auto)
 qed
next
 assume "p6(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"
 with type show ?thesis
  by (clarsimp simp: TypeOK_def p6_def, auto)
next
 assume "p7(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"
 with type show ?thesis
  by (clarsimp simp: TypeOK_def p7_def, auto)
next
 assume "p8(self,unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"
 with type show ?thesis
```

```
  by (clarsimp simp: TypeOK_def p8_def, auto)
qed

theorem InvImpliesMutex:
 assumes inv: "Inv(unread,max,flag,pc,num,nxt)"
 shows "MutualExclusion(pc)"
proof (clarsimp simp: MutualExclusion_def)
 fix i j
 assume hyps: "i ∈ ProcSet" "j ∈ ProcSet" "i ≠ j" "pc[i] = ”p7”” "pc[j] = ”p7”"
 with inv have "IInv(i, unread, max, flag, pc, num, nxt)"
  and "IInv(j, unread, max, flag, pc, num, nxt)"
  by (auto simp: Inv_def)
 with hyps have "After(i,j,unread,max,flag,pc,num,nxt)"
  and "After(j,i,unread,max,flag,pc,num,nxt)"
  by (auto simp add: IInv_def)
 with hyps show FALSE
  by (auto simp add: After_def)
qed
```

B.1.2. System Invariant proof

```
theorem InitImpliesInv:
 assumes init: "Init(unread,max,flag,pc,num,nxt)"
 shows "Inv(unread,max,flag,pc,num,nxt)"
proof -
 from init have "TypeOK(unread,max,flag,pc,num,nxt)"
  by (rule InitImpliesTypeOK)
 with init show ?thesis
  by (auto simp: Inv_def IInv_def Init_def)
qed

theorem InvInvariant:
 assumes inv: "Inv(unread,max,flag,pc,num,nxt)"
    and nxt: "Next(unread,max,flag,pc,num,nxt,unread',max',flag',pc',num',nxt')"
 shows "Inv(unread',max',flag',pc',num',nxt')"
using assms unfolding Next_def
proof auto
 fix self
 assume self: "self ∈ ProcSet"
   and p: "p(self, unread, max, flag, pc, num, nxt, unread', max', flag', pc', num', nxt')"
 with inv show ?thesis
 proof (auto simp: Inv_def elim: TypeOKInvariant)
  fix i
  assume type: "TypeOK(unread, max, flag, pc, num, nxt)"
    and iinv: "∀ j ∈ ProcSet: IInv(j, unread, max, flag, pc, num, nxt)"
    and i: "i ∈ ProcSet"
  — auxiliary definition that is used in several places of the proof below
  def after ≡ "λk. pc[k] = ”p1” ∨
```

```
      pc[k] = "p2" ∧ (i ∈ unread[k] ∨ num[i] ≤ max[k]) ∨
      pc[k] = "p3" ∧ num[i] ≤ max[k] ∨
      (pc[k] = "p4" ∨ pc[k] = "p5" ∨ pc[k] = "p6") ∧
      GG(i, k, num) ∧ (pc[k] = "p5" ∨ pc[k] = "p6" ⇒ i ∈ unread[k])"
  from iinv i have iinvi: "IInv(i, unread, max, flag, pc, num, nxt)" ..
```

— iinv3 and iinv5: particular parts of the invariant, taken to the meta-level for then being instantiated with the proper variables, to ease the work of the classical reasoner.

```
  from iinv
  have iinv3: "⋀i j.
   ⟦pc[i] ∈ {"p5", "p6"}; i ∈ ProcSet; j ∈ ProcSet \ unread[i] \ {i}⟧
   ⟹ After(j, i, unread, max, flag, pc, num, nxt)"
  proof -
   fix i j
   assume pci: "pc[i] ∈ {"p5", "p6"}"
    and i: "i ∈ ProcSet" and j:"j ∈ ProcSet \ unread[i] \ {i}"
   from iinv i have iinvi: "IInv(i, unread, max, flag, pc, num, nxt)" ..
   hence "pc[i] ∈ {"p5", "p6"} ⇒
    (∀j ∈ ProcSet \ unread[i] \ {i} :
    After(j, i, unread, max, flag, pc, num, nxt))"
    unfolding IInv_def by auto
   with pci i j
   show "After(j, i, unread, max, flag, pc, num, nxt)"
    by auto
  qed

  from iinv
  have iinv5: "⋀i j.
   ⟦pc[i] ∈ {"p7", "p8"}; i ∈ ProcSet; j ∈ ProcSet \ {i}⟧
   ⟹ After(j, i, unread, max, flag, pc, num, nxt)"
  proof -
   fix i j
   assume pci: "pc[i] ∈ {"p7", "p8"}"
    and i: "i ∈ ProcSet" and j:"j ∈ ProcSet \ {i}"
   from iinv i have iinvi: "IInv(i, unread, max, flag, pc, num, nxt)" ..
   hence "pc[i] ∈ {"p7", "p8"} ⇒
    (∀j ∈ ProcSet \ {i} :
    After(j, i, unread, max, flag, pc, num, nxt))"
    unfolding IInv_def by auto
   with pci i j
   show "After(j, i, unread, max, flag, pc, num, nxt)"
    by auto
  qed
```

— This also is an instantiation of a type invariant, since auto can't resolve it in a reasonable time.

```
  from type i
  have nxti: "pc[i] = "p6" ⇒ nxt[i] ∈ ProcSet ∧ nxt[i] ≠ i"
   by (auto simp: TypeOK_def)
```

```
from p show "IInv(i, unread', max', flag', pc', num', nxt')"
proof (auto simp: p_def)
 assume p1: "p1(self, unread, max, flag, pc, num, nxt, unread', max', flag', pc', num', nxt')"
 show ?thesis
 proof (cases "self = i")
  case True
  with p1 type iinvi i show ?thesis
   by (clarsimp simp: TypeOK_def IInv_def p1_def)
 next
  assume selfi: "self ≠ i"
  with p1 type iinvi self i
  have 1: "(num'[i] = 0) = (pc'[i] ∈ {"p1", "p2", "p3"})"
   by (clarsimp simp: TypeOK_def IInv_def p1_def)
  from selfi p1 type iinvi self i
  have 2: "flag'[i] = (pc'[i] ∈ {"p2", "p3", "p4"})"
   by (clarsimp simp: TypeOK_def IInv_def p1_def)
  have 3: "pc'[i] ∈ {"p5", "p6"} ⇒
        (∀ j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
  proof (rule+)
   fix j
   assume pc': "pc'[i] ∈ {"p5","p6"}" and j: "j ∈ (ProcSet \ unread'[i]) \ {i}"
   show "After(j,i,unread',max',flag',pc',num',nxt')"
   proof (cases "self = j")
    case True with selfi p1 type iinvi self i pc' show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def p1_def nat_gt0_not0)

     assume "j ≠ i" "pc[i] = "p5" ∨ pc[i] = "p6""
     thus "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
     by auto
    qed
   next
    case False with selfi p1 type iinvi self i pc' j show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def p1_def nat_gt0_not0)

     assume "pc[i] = "p5" ∨ pc[i] = "p6""
     hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3"" by auto
     assume "∀ k ∈ ProcSet \ unread[i] \ {i}:
           pc[k] = "p1" ∨
           pc[k] = "p2" ∧ (i ∈ unread[k] ∨ num[i] ≤ max[k]) ∨
           pc[k] = "p3" ∧ num[i] ≤ max[k] ∨
           (pc[k] = "p4" ∨ pc[k] = "p5" ∨ pc[k] = "p6") ∧
           GG(i, k, num) ∧ (pc[k] = "p5" ∨ pc[k] = "p6" ⇒ i ∈ unread[k])"
     hence aft: "∀ k ∈ ProcSet \ unread[i] \ {i}: after(k)" unfolding after_def .
     assume "j ∈ ProcSet" "j ∉ unread[i]" "j ≠ i"
     with aft have "after(j)" by blast
     hence "pc[j] = "p1" ∨
        pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
```

```
          pc[j] = ”p3” ∧ num[i] ≤ max[j] ∨
          (pc[j] = ”p4” ∨ pc[j] = ”p5” ∨ pc[j] = ”p6”) ∧
          GG(i, j, num) ∧ (pc[j] = ”p5” ∨ pc[j] = ”p6” ⇒ i ∈ unread[j])”
       unfolding after_def .
     with ii1
     show “pc[i] ≠ ”p1” ∧ pc[i] ≠ ”p2” ∧ pc[i] ≠ ”p3”
         ∧ (pc[j] = ”p1” ∨
           pc[j] = ”p2” ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
           pc[j] = ”p3” ∧ num[i] ≤ max[j] ∨
           (pc[j] = ”p4” ∨ pc[j] = ”p5” ∨ pc[j] = ”p6”) ∧
           GG(i, j, num) ∧ (pc[j] = ”p5” ∨ pc[j] = ”p6” ⇒ i ∈ unread[j]))”
       by simp
   qed
  qed
qed
have 4: “pc’[i] = ”p6”
       ∧ ( (pc’[nxt’[i]] = ”p2”) ∧ i ∉ unread’[nxt’[i]]
         ∨ (pc’[nxt’[i]] = ”p3”))
       ⇒ max’[nxt’[i]] ≥ num’[i]”
proof (cases “self = nxt[i]”)
  case True
  with selfi p1 type iinvi self i show ?thesis
    by (clarsimp simp: TypeOK_def IInv_def p1_def)
next
  case False
  with type self i have “pc[i] = ”p6” ⇒ nxt[i] ∈ ProcSet”
    by (auto simp: TypeOK_def)
  with False selfi p1 type iinvi self i show ?thesis
    by (clarsimp simp: TypeOK_def IInv_def p1_def)
qed
have 5: “(pc’[i] ∈ {”p7”, ”p8”}) ⇒ (∀ j ∈ P \ {i} : After(j,i,unread’,max’,flag’,pc’,num’,nxt’))”
proof (rule+)
  fix j
  assume pc’: “pc’[i] ∈ {”p7”,”p8”}” and j: “j ∈ ProcSet \ {i}”
  show “After(j,i,unread’,max’,flag’,pc’,num’,nxt’)”
  proof (cases “self = j”)
    case True with selfi p1 type iinvi self i pc’ show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def p1_def nat_gt0_not0)

      assume “j ≠ i” “pc[i] = ”p7” ∨ pc[i] = ”p8””
      thus “pc[i] ≠ ”p1” ∧ pc[i] ≠ ”p2” ∧ pc[i] ≠ ”p3””
        by auto
    qed
  next
    case False with selfi p1 type iinvi self i pc’ j show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def p1_def nat_gt0_not0)

      assume “pc[i] = ”p7” ∨ pc[i] = ”p8””
```

```
      hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3"" by auto
      assume "∀k ∈ ProcSet \ {i} :
              pc[k] = "p1" ∨
              pc[k] = "p2" ∧ (i ∈ unread[k] ∨ num[i] ≤ max[k]) ∨
              pc[k] = "p3" ∧ num[i] ≤ max[k] ∨
              (pc[k] = "p4" ∨ pc[k] = "p5" ∨ pc[k] = "p6") ∧
              GG(i, k, num) ∧ (pc[k] = "p5" ∨ pc[k] = "p6" ⇒ i ∈ unread[k])"
      hence aft: "∀k ∈ ProcSet \ {i} : after(k)" unfolding after_def .
      with j have "after(j)" by blast
      hence "pc[j] = "p1" ∨
            pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
            pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
            (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
            GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
       unfolding after_def .
      with ii1
      show "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3"
          ∧ (pc[j] = "p1" ∨
            pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
            pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
            (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
            GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j]))"
       by simp
     qed
    qed
   qed
   from 1 2 3 4 5 show ?thesis
    unfolding IInv_def by blast
  qed
next
   assume p2: "p2(self, unread, max, flag, pc, num, nxt, unread', max', flag', pc', num', nxt')"
   show ?thesis
   proof (cases "self = i")
    assume selfi: "self = i"
    show ?thesis
    proof (cases "unread[self] = {}")
     case True
     with selfi p2 type iinvi i show ?thesis
      by (clarsimp simp: TypeOK_def IInv_def p2_def)
    next
     case False
     from type have "isAFcn(pc)"

      by (force simp add: TypeOK_def)
     with False p2 obtain k where
      k: "pc[self] = "p2"" "k ∈ unread[self]"
        "unread' = [unread EXCEPT ![self] = unread[self] \ {k}]"
        "max' = (IF num[k] > max[self] THEN [max EXCEPT ![self] = num[k]] ELSE max)"
```

```
      "pc' = pc" "num' = num" "flag' = flag" "nxt' = nxt"
    by (auto simp: p2_def)
  with selfi type iinvi i show ?thesis
    by (clarsimp simp: TypeOK_def IInv_def)
 qed
next
 assume selfi: "self ≠ i"
 show ?thesis
 proof (cases "unread[self] = {}")
  assume empty: "unread[self] = {}"
  with selfi p2 type iinvi self i
  have 1: "(num'[i] = 0) = (pc'[i] ∈ {"p1", "p2", "p3"})"
   by (clarsimp simp: TypeOK_def IInv_def p2_def)
  from empty selfi p2 type iinvi self i
  have 2: "flag'[i] = (pc'[i] ∈ {"p2", "p3", "p4"})"
   by (clarsimp simp: TypeOK_def IInv_def p2_def)
  have 3: "pc'[i] ∈ {"p5", "p6"} ⇒
       (∀j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
  proof (rule+)
   fix j
   assume pc': "pc'[i] ∈ {"p5","p6"}" and j: "j ∈ (ProcSet \ unread'[i]) \ {i}"
   from j type have mx: "pc[j] = "p2" ⇒ max[j] ∈ Nat"
    by (auto simp: TypeOK_def)
   show "After(j,i,unread',max',flag',pc',num',nxt')"
   proof (cases "self = j")
    case True with empty selfi p2 type iinvi self i pc' j mx show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def p2_def nat_gt0_not0)

     assume "pc[i] = "p5" ∨ pc[i] = "p6""
     hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
      by auto
     assume "j ∈ ProcSet" "j ≠ i" "j ∉ unread[i]" "pc[j] = "p2""
          "∀i: i ∉ unread[j]"
          "∀j ∈ ProcSet \ unread[i] \ {i}:
            pc[j] = "p1" ∨
            pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
            pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
            (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
            GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
     hence "num[i] ≤ max[j]" by auto
     with ii1 show "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3" ∧ num[i] ≤ max[j]"
      by simp
    qed
   next
    case False with empty selfi p2 type iinvi self i pc' j show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def p2_def nat_gt0_not0)

     assume "pc[i] = "p5" ∨ pc[i] = "p6""
```

```
    hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
     by auto
    assume "∀ j ∈ ProcSet \ unread[i] \ {i}:
            pc[j] = "p1" ∨
            pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
            pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
            (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
            GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
    hence aft: "∀ j ∈ ProcSet \ unread[i] \ {i}: after(j)"
     unfolding after_def .
    assume "j ∈ ProcSet" "j ≠ i" "j ∉ unread[i]"
    with aft have "after(j)" by blast
    with ii1
    show "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3"
        ∧ (pc[j] = "p1" ∨
           pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
           pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
           (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
           GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j]))"
     by (simp add: after_def)
   qed
  qed
qed
have 4: "pc'[i] = "p6"
      ∧ (  (pc'[nxt'[i]] = "p2") ∧ i ∉ unread'[nxt'[i]]
        ∨ (pc'[nxt'[i]] = "p3"))
      ⇒ max'[nxt'[i]] ≥ num'[i]"
proof (cases "self = nxt[i]")
  case True
  with empty selfi p2 type iinvi self i show ?thesis
    by (clarsimp simp: TypeOK_def IInv_def p2_def, auto)
next
  case False
  with type self i have "pc[i] = "p6" ⇒ nxt[i] ∈ ProcSet"
    by (auto simp: TypeOK_def)
  with False empty selfi p2 type iinvi self i show ?thesis
    by (clarsimp simp: TypeOK_def IInv_def p2_def)
qed
have 5: "(pc'[i] ∈ {"p7", "p8"}) ⇒ (∀ j ∈ P \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
proof (rule+)
  fix j
  assume pc': "pc'[i] ∈ {"p7","p8"}" and j: "j ∈ ProcSet \ {i}"
  from j type have mx: "pc[j] = "p2" ⇒ max[j] ∈ Nat"
    by (auto simp: TypeOK_def)
  show "After(j,i,unread',max',flag',pc',num',nxt')"
  proof (cases "self = j")
    case True
    with empty selfi p2 type iinvi i pc' j mx show ?thesis
```

```
  proof (clarsimp simp: TypeOK_def IInv_def After_def p2_def nat_gt0_not0)

    assume "pc[i] = "p7" ∨ pc[i] = "p8""
    hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
      by auto
    assume "j ∈ ProcSet" "j ≠ i" "pc[j] = "p2""
         "∀ i: i ∉ unread[j]"
         "∀ j ∈ ProcSet \ {i}:
            pc[j] = "p1" ∨
            pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
            pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
            (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
            GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
    hence "num[i] ≤ max[j]" by auto
    with ii1 show "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3" ∧ num[i] ≤ max[j]"
      by simp
  qed
 next
  case False
  with empty selfi p2 type iinvi i pc' j show ?thesis
  proof (clarsimp simp: TypeOK_def IInv_def After_def p2_def nat_gt0_not0)

    assume "pc[i] = "p7" ∨ pc[i] = "p8""
    hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
      by auto
    assume "∀ j ∈ ProcSet \ {i}:
            pc[j] = "p1" ∨
            pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
            pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
            (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
            GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
    hence aft: "∀ j ∈ ProcSet \ {i}: after(j)"
      unfolding after_def .
    assume "j ∈ ProcSet" "j ≠ i"
    with aft have "after(j)" by blast
    with ii1
    show "pc[i] ≠ "p1" ∧
          pc[i] ≠ "p2" ∧
          pc[i] ≠ "p3" ∧
          (pc[j] = "p1" ∨
           pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
           pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
           (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
           GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j]))"
      by (simp add: after_def)
  qed
 qed
qed
```

```
  from 1 2 3 4 5 show ?thesis
    unfolding IInv_def by blast
next
  assume nempty: "unread[self] ≠ {}"
  from type have "isAFcn(pc)"

    by (force simp: TypeOK_def)
  with nempty p2 obtain k where
    k: "pc[self] = ”p2”" "k ∈ unread[self]"
      "unread' = [unread EXCEPT ![self] = unread[self] \ {k}]"
      "max' = (IF num[k] > max[self] THEN [max EXCEPT ![self] = num[k]] ELSE max)"
      "pc' = pc" "num' = num" "flag' = flag" "nxt' = nxt"
    by (auto simp: p2_def)
  with type self have kproc: "k ∈ ProcSet"
    by (auto simp: TypeOK_def)
  from k selfi type iinvi self i
  have 1: "(num'[i] = 0) = (pc'[i] ∈ {”p1”, ”p2”, ”p3”})"
    by (clarsimp simp: TypeOK_def IInv_def)
  from k selfi type iinvi self i
  have 2: "flag'[i] = (pc'[i] ∈ {”p2”, ”p3”, ”p4”})"
    by (clarsimp simp: TypeOK_def IInv_def)
  have 3: "pc'[i] ∈ {”p5”, ”p6”} ⇒
        (∀j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
  proof (rule+)
    fix j
    assume pc': "pc'[i] ∈ {”p5”,”p6”}" and j: "j ∈ (ProcSet \ unread'[i]) \ {i}"
    from j type have mx: "pc[j] = ”p2” ⇒ max[j] ∈ Nat"
      by (auto simp: TypeOK_def)
    show "After(j,i,unread',max',flag',pc',num',nxt')"
    proof (cases "self = j")
      assume selfj: "self = j"
      show ?thesis
      proof (cases "max[j] < num[k]")
        assume less: "max[j] < num[k]"
        show ?thesis
        proof (cases "i=k")
          case True
          with k selfi selfj less type iinvi self i pc' j mx show ?thesis
          proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0)
            assume "pc[k] = ”p5” ∨ pc[k] = ”p6”"
            thus "pc[k] ≠ ”p1” ∧ pc[k] ≠ ”p2” ∧ pc[k] ≠ ”p3”"
              by auto
          qed
        next
          case False
          with k selfi selfj less type iinvi self i pc' j mx show ?thesis
          proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0)
            assume "pc[i] = ”p5” ∨ pc[i] = ”p6”"
```

```
    hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
      by auto
    assume "pc[j] = "p2"" "j ∈ ProcSet" "j ∉ unread[i]" "j ≠ i"
        "∀j ∈ ProcSet \ unread[i] \ {i} :
          pc[j] = "p1" ∨
          pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
          pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
          (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
          GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
    hence ii2: "i ∈ unread[j] ∨ num[i] ≤ max[j]"
      by auto
    assume "max[j] < num[k]" "num ∈ [ProcSet → Nat]" "i ∈ ProcSet"
        "max[j] ∈ Nat"
    with ii2 kproc have "i ∈ unread[j] ∨ num[i] ≤ num[k]"
      by (auto dest: nat_leq_less_trans)
    with ii1
    show "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3"
       ∧ (i ∈ unread[j] ∨ num[i] ≤ num[k])"
      by simp
  qed
 qed
next
 assume nless: "¬(max[j] < num[k])"
 show ?thesis
 proof (cases "i=k")
  case True
  with k selfi selfj nless type iinvi self i pc' j mx show ?thesis
  proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0 condElse nat_not_less)
   assume "pc[k] = "p5" ∨ pc[k] = "p6""
   thus "pc[k] ≠ "p1" ∧ pc[k] ≠ "p2" ∧ pc[k] ≠ "p3""
    by auto
  qed
 next
  case False
  with k selfi selfj nless type iinvi self i pc' j mx show ?thesis
  proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0 condElse)
   assume "pc[i] = "p5" ∨ pc[i] = "p6""
   hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
    by auto
   assume "j ∈ ProcSet" "j ∉ unread[i]" "j ≠ i" "pc[j] = "p2""
       "∀j ∈ ProcSet \ unread[i] \ {i} :
         pc[j] = "p1" ∨
         pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
         pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
         (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
         GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
   hence "i ∈ unread[j] ∨ num[i] ≤ max[j]"
    by auto
```

```
      with ii1 show "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3"
                  ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j])"
        by simp
      qed
    qed
  qed
next
  assume selfj: "self ≠ j"
  show ?thesis
  proof (cases "max[self] < num[k]")
    case True
    with k selfi selfj type iinvi self i pc' j show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0)
      assume "pc[i] = "p5" ∨ pc[i] = "p6""
      hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
        by auto
      assume "∀ j ∈ ProcSet \ unread[i] \ {i} :
              pc[j] = "p1" ∨
              pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
              pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
              (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
              GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
      hence aft: "∀ j ∈ ProcSet \ unread[i] \ {i} : after(j)"
        unfolding after_def .
      assume "j ∈ ProcSet" "j ∉ unread[i]" "j ≠ i"
      with aft have "after(j)" by blast
      with ii1 show
        "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3" ∧
        (pc[j] = "p1" ∨
         pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
         pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
         (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
         GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j]))"
        by (simp add: after_def)
    qed
  next
    case False
    with k selfi selfj type iinvi self i pc' j show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0 condElse)
      assume "pc[i] = "p5" ∨ pc[i] = "p6""
      hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
        by auto
      assume "∀ j ∈ ProcSet \ unread[i] \ {i} :
              pc[j] = "p1" ∨
              pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
              pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
              (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
              GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
```

```
      hence aft: "∀j ∈ ProcSet \ unread[i] \ {i} : after(j)"
       unfolding after_def .
      assume "j ∈ ProcSet" "j ∉ unread[i]" "j ≠ i"
      with aft have "after(j)" by blast
      with ii1 show
       "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3" ∧
       (pc[j] = "p1" ∨
        pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
        pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
        (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
        GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j]))"
       by (simp add: after_def)
     qed
    qed
  qed
qed
have 4: "pc'[i] = "p6"
      ∧ ( (pc'[nxt'[i]] = "p2") ∧ i ∉ unread'[nxt'[i]]
        ∨ (pc'[nxt'[i]] = "p3"))
      ⇒ max'[nxt'[i]] ≥ num'[i]"
proof (cases "self = nxt[i]")
  assume nxt: "self = nxt[i]"
  from type k self have mx: "max[self] ∈ Nat"
   by (auto simp: TypeOK_def)
  show ?thesis
  proof (cases "max[self] < num[k]")
   case True
   with k selfi type iinvi self i nxt mx show ?thesis
   proof (clarsimp simp: TypeOK_def IInv_def, cases "i=k", simp, simp)
    assume "num ∈ [ProcSet → Nat]" "num[i] ≤ max[nxt[i]]" "max[nxt[i]] < num[k]"
    with mx kproc i nxt show "num[i] ≤ num[k]"
     by (auto dest: nat_leq_less_trans)
   qed
  next
   case False
   with k selfi type iinvi self i nxt mx show ?thesis
    by (clarsimp simp: TypeOK_def IInv_def condElse nat_not_less)
  qed
next
  assume nxt: "self ≠ nxt[i]"
  from type self i have pc6: "pc[i] = "p6" ⇒ nxt[i] ∈ ProcSet"
   by (auto simp: TypeOK_def)
  show ?thesis
  proof (cases "max[self] < num[k]")
   case True
   with k selfi type iinvi self i nxt pc6 show ?thesis
    by (clarsimp simp: TypeOK_def IInv_def)
  next
```

```
    case False
    with k selfi type iinvi self i nxt pc6 show ?thesis
     by (clarsimp simp: TypeOK_def IInv_def condElse)
  qed
qed
have 5: "(pc'[i] ∈ {"p7", "p8"}) ⇒ (∀ j ∈ P \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
proof (rule+)
  fix j
  assume pc': "pc'[i] ∈ {"p7","p8"}" and j: "j ∈ ProcSet \ {i}"
  from j type have mx: "pc[j] = "p2" ⇒ max[j] ∈ Nat"
   by (auto simp: TypeOK_def)
  show "After(j,i,unread',max',flag',pc',num',nxt')"
  proof (cases "self = j")
    assume selfj: "self = j"
    show ?thesis
    proof (cases "max[j] < num[k]")
      assume less: "max[j] < num[k]"
      show ?thesis
      proof (cases "i=k")
        case True
        with k selfi selfj less type iinvi i pc' j mx show ?thesis
        proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0)
          assume "pc[k] = "p7" ∨ pc[k] = "p8""
          thus "pc[k] ≠ "p1" ∧ pc[k] ≠ "p2" ∧ pc[k] ≠ "p3""
            by auto
        qed
      next
        case False
        with k selfi selfj less type iinvi i pc' j mx show ?thesis
        proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0)
          assume "pc[i] = "p7" ∨ pc[i] = "p8""
          hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
            by auto
          assume "j ∈ ProcSet" "j ≠ i" "pc[j] = "p2""
                 "∀ j ∈ ProcSet \ {i} :
                   pc[j] = "p1" ∨
                   pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
                   pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
                   (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
                   GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
          hence ii2: "i ∈ unread[j] ∨ num[i] ≤ max[j]"
            by auto
          assume "max[j] < num[k]" "num ∈ [ProcSet → Nat]" "max[j] ∈ Nat"
          with i kproc ii2 have "i ∈ unread[j] ∨ num[i] ≤ num[k]"
            by (auto dest: nat_leq_less_trans)
          with ii1 show "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3"
                       ∧ (i ∈ unread[j] ∨ num[i] ≤ num[k])"
            by simp
```

```
    qed
   qed
  next
   assume nless: "¬(max[j] < num[k])"
   show ?thesis
   proof (cases "i=k")
    case True
    with k selfi selfj nless type iinvi i pc' j mx show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0 condElse nat_not_less)
     assume "pc[k] = "p7" ∨ pc[k] = "p8""
     thus "pc[k] ≠ "p1" ∧ pc[k] ≠ "p2" ∧ pc[k] ≠ "p3""
      by auto
    qed
   next
    case False
    with k selfi selfj nless type iinvi i pc' j mx show ?thesis
    proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0 condElse)
     assume "pc[i] = "p7" ∨ pc[i] = "p8""
     hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
      by auto
     assume "j ∈ ProcSet" "j ≠ i" "pc[j] = "p2""
          "∀ j ∈ ProcSet \ {i} :
            pc[j] = "p1" ∨
            pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
            pc[j] = "p3" ∧ num[i] ≤ max[j] ∨
            (pc[j] = "p4" ∨ pc[j] = "p5" ∨ pc[j] = "p6") ∧
            GG(i, j, num) ∧ (pc[j] = "p5" ∨ pc[j] = "p6" ⇒ i ∈ unread[j])"
     hence "i ∈ unread[j] ∨ num[i] ≤ max[j]"
      by auto
     with ii1 show "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3"
                 ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j])"
      by simp
    qed
   qed
  qed
 next
  assume selfj: "self ≠ j"
  show ?thesis
  proof (cases "max[self] < num[k]")
   case True
   with k selfi selfj type iinvi self i pc' j show ?thesis
   proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0)
    assume "pc[i] = "p7" ∨ pc[i] = "p8""
    hence ii1: "pc[i] ≠ "p1" ∧ pc[i] ≠ "p2" ∧ pc[i] ≠ "p3""
     by auto
    assume "∀ j ∈ ProcSet \ {i} :
          pc[j] = "p1" ∨
          pc[j] = "p2" ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
```

```
            pc[j] = ”p3” ∧ num[i] ≤ max[j] ∨
            (pc[j] = ”p4” ∨ pc[j] = ”p5” ∨ pc[j] = ”p6”) ∧
            GG(i, j, num) ∧ (pc[j] = ”p5” ∨ pc[j] = ”p6” ⇒ i ∈ unread[j])”
      hence aft: “∀ j ∈ ProcSet \ {i} : after(j)”
       unfolding after_def .
      assume “j ∈ ProcSet” “j ≠ i”
      with aft have “after(j)” by blast
      with ii1 show
       “pc[i] ≠ ”p1” ∧ pc[i] ≠ ”p2” ∧ pc[i] ≠ ”p3” ∧
       (pc[j] = ”p1” ∨
        pc[j] = ”p2” ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
        pc[j] = ”p3” ∧ num[i] ≤ max[j] ∨
        (pc[j] = ”p4” ∨ pc[j] = ”p5” ∨ pc[j] = ”p6”) ∧
        GG(i, j, num) ∧ (pc[j] = ”p5” ∨ pc[j] = ”p6” ⇒ i ∈ unread[j]))”
       by (simp add: after_def)
     qed
    next
     case False
     with k selfi selfj type iinvi self i pc’ j show ?thesis
     proof (clarsimp simp: TypeOK_def IInv_def After_def nat_gt0_not0 condElse)
      assume “pc[i] = ”p7” ∨ pc[i] = ”p8””
      hence ii1: “pc[i] ≠ ”p1” ∧ pc[i] ≠ ”p2” ∧ pc[i] ≠ ”p3””
       by auto
      assume “∀ j ∈ ProcSet \ {i} :
            pc[j] = ”p1” ∨
            pc[j] = ”p2” ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
            pc[j] = ”p3” ∧ num[i] ≤ max[j] ∨
            (pc[j] = ”p4” ∨ pc[j] = ”p5” ∨ pc[j] = ”p6”) ∧
            GG(i, j, num) ∧ (pc[j] = ”p5” ∨ pc[j] = ”p6” ⇒ i ∈ unread[j])”
      hence aft: “∀ j ∈ ProcSet \ {i} : after(j)”
       unfolding after_def .
      assume “j ∈ ProcSet” “j ≠ i”
      with aft have “after(j)” by blast
      with ii1 show
       “pc[i] ≠ ”p1” ∧ pc[i] ≠ ”p2” ∧ pc[i] ≠ ”p3” ∧
       (pc[j] = ”p1” ∨
        pc[j] = ”p2” ∧ (i ∈ unread[j] ∨ num[i] ≤ max[j]) ∨
        pc[j] = ”p3” ∧ num[i] ≤ max[j] ∨
        (pc[j] = ”p4” ∨ pc[j] = ”p5” ∨ pc[j] = ”p6”) ∧
        GG(i, j, num) ∧ (pc[j] = ”p5” ∨ pc[j] = ”p6” ⇒ i ∈ unread[j]))”
       by (simp add: after_def)
     qed
    qed
   qed
  qed
  from 1 2 3 4 5 show ?thesis
   unfolding IInv_def by blast
qed
```

```
  qed
 next
  assume p3: "p3(self, unread, max, flag, pc, num, nxt, unread', max', flag', pc', num', nxt')"
  show ?thesis
  proof (cases "self = i")
   case True
   with p3 type iinvi i show ?thesis
    by (clarsimp simp: TypeOK_def IInv_def p3_def, auto)
  next
   assume selfi: "self ≠ i"
   from selfi p3 type iinvi self i
   have 1: "(num'[i] = 0) = (pc'[i] ∈ {"p1", "p2", "p3"})"
    and 2: "flag'[i] = (pc'[i] ∈ {"p2", "p3", "p4"})"
    by (clarsimp simp: TypeOK_def IInv_def p3_def)+
   have 3: "pc'[i] ∈ {"p5", "p6"} ⇒
    (∀j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
   proof (rule+)
    fix j
    assume pc': "pc'[i] ∈ {"p5","p6"}" and j: "j ∈ (ProcSet \ unread'[i]) \ {i}"
    show "After(j,i,unread',max',flag',pc',num',nxt')"
    proof (cases "self = j")
     case True with selfi type p3 self i j iinv3[of i j] pc' show ?thesis
      by (clarsimp simp: TypeOK_def p3_def After_def GG_def, auto)

     next
      case False with selfi p3 type self i j iinv3[of i j] pc'
      show ?thesis
       by (clarsimp simp: TypeOK_def p3_def After_def GG_def)
     qed
    qed
   have 4: "pc'[i] = "p6"
        ∧ ( (pc'[nxt'[i]] = "p2") ∧ i ∉ unread'[nxt'[i]]
          ∨ (pc'[nxt'[i]] = "p3"))
        ⇒ max'[nxt'[i]] ≥ num'[i]"
   proof (cases "self = nxt[i]")
    case True
    with selfi p3 type self i show ?thesis
     by (clarsimp simp: TypeOK_def p3_def)
   next
    case False
    with selfi p3 type iinvi self i nxti show ?thesis
     by (clarsimp simp: TypeOK_def IInv_def p3_def)
   qed
   have 5: "(pc'[i] ∈ {"p7", "p8"}) ⇒ (∀j ∈ P \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
   proof (rule+)
    fix j
    assume pc': "pc'[i] ∈ {"p7","p8"}" and j: "j ∈ ProcSet \ {i}"
    show "After(j,i,unread',max',flag',pc',num',nxt')"
```

```
    proof (cases "self = j")
      case True
      with selfi type p3 self i j pc' iinv5[of i j] show ?thesis
        by (clarsimp simp: TypeOK_def p3_def After_def GG_def, auto)
    next
      case False
      with selfi p3 type self i j pc' iinv5[of i j] show ?thesis
        by (clarsimp simp: TypeOK_def p3_def After_def GG_def)
    qed
  qed
  from 1 2 3 4 5 show ?thesis
    unfolding IInv_def by blast
qed
next
  assume p4: "p4(self, unread, max, flag, pc, num, nxt, unread', max', flag', pc', num', nxt')"
  show ?thesis
  proof (cases "self = i")
    case True
    with p4 type iinvi i show ?thesis
      by (clarsimp simp: TypeOK_def IInv_def p4_def)
  next
    assume selfi: "self ≠ i"
    with p4 type iinvi self i
    have 1: "(num'[i] = 0) = (pc'[i] ∈ {"p1", "p2", "p3"})"
      and 2: "flag'[i] = (pc'[i] ∈ {"p2", "p3", "p4"})"
      by (clarsimp simp: TypeOK_def IInv_def p4_def)+
    have 3: "pc'[i] ∈ {"p5", "p6"} ⇒
          (∀ j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
    proof (rule+)
      fix j
      assume pc': "pc'[i] ∈ {"p5","p6"}" and j: "j ∈ (ProcSet \ unread'[i]) \ {i}"
      with selfi type p4 self i iinv3[of i j]
      show "After(j,i,unread',max',flag',pc',num',nxt')"
        unfolding TypeOK_def p4_def After_def GG_def
        by(cases "self = j", clarsimp+)
    qed
    have 4: "pc'[i] = "p6"
          ∧ ( (pc'[nxt'[i]] = "p2") ∧ i ∉ unread'[nxt'[i]]
            ∨ (pc'[nxt'[i]] = "p3"))
          ⇒ max'[nxt'[i]] ≥ num'[i]"
    proof (cases "self = nxt[i]")
      case True
      with selfi p4 type self i show ?thesis
        by (clarsimp simp: TypeOK_def p4_def)
    next
      case False
      with selfi p4 type iinvi self i nxti show ?thesis
        by (clarsimp simp: TypeOK_def IInv_def p4_def)
```

```
    qed
    have 5: "(pc'[i] ∈ {"p7", "p8"}) ⇒ (∀ j ∈ P \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
    proof (rule+)
      fix j
      assume pc': "pc'[i] ∈ {"p7","p8"}" and j: "j ∈ ProcSet \ {i}"
      with selfi type p4 self i j pc' iinv5[of i j]
      show "After(j,i,unread',max',flag',pc',num',nxt')"
        apply (cases "self = j")
        by (clarsimp simp: TypeOK_def p4_def After_def)+
    qed
    from 1 2 3 4 5 show ?thesis
      unfolding IInv_def by blast
  qed
next
  assume p5: "p5(self, unread, max, flag, pc, num, nxt, unread', max', flag', pc', num', nxt')"
  show ?thesis
  proof (cases "unread[self] = {}")
    assume empty: "unread[self] = {}"
    from p5 type iinvi self i empty
    have 1: "(num'[i] = 0) = (pc'[i] ∈ {"p1", "p2", "p3"})"
      unfolding TypeOK_def IInv_def p5_def
      by(cases "self = i", clarsimp+)
    from p5 type iinvi self i empty
    have 2: "flag'[i] = (pc'[i] ∈ {"p2", "p3", "p4"})"
      unfolding TypeOK_def IInv_def p5_def
      by(cases "self = i", clarsimp+)
    have 3: "pc'[i] ∈ {"p5", "p6"} ⇒
     (∀ j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
    proof (rule+)
      fix j
      assume pc': "pc'[i] ∈ {"p5","p6"}" and j: "j ∈ (ProcSet \ unread'[i]) \ {i}"
      with empty type p5 self i j iinv3[of i j]
      show "After(j,i,unread',max',flag',pc',num',nxt')"
        unfolding TypeOK_def After_def GG_def p5_def
        apply(cases "self = j", force, clarsimp)
        by(cases "self = i", simp+)
    qed
    have 4: "pc'[i] = "p6"
     ∧ (  (pc'[nxt'[i]] = "p2") ∧ i ∉ unread'[nxt'[i]]
     ∨ (pc'[nxt'[i]] = "p3"))
     ⇒ max'[nxt'[i]] ≥ num'[i]"
    proof -
      from empty p5 type iinvi self i show ?thesis
        unfolding TypeOK_def IInv_def p5_def
        apply(cases "self = i")
          apply clarsimp
          apply(cases "self = nxt[i]", clarsimp, clarsimp simp: nxti)
        done
```

```
  qed
  have 5: "(pc'[i] ∈ {"p7", "p8"}) ⇒ (∀j ∈ P \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
  proof (rule+)
   fix j
   assume pc': "pc'[i] ∈ {"p7","p8"}" and j: "j ∈ ProcSet \ {i}"
   show "After(j,i,unread',max',flag',pc',num',nxt')"
   proof (cases "self = i")
    case True
    with empty type p5 self i j iinv3[of i j]
    show ?thesis
     by (clarsimp simp: IInv_def TypeOK_def p5_def After_def, force)
   next
    from empty
    have unreadj: "j ∉ unread[self]" by auto
    assume selfi: "self ≠ i"
    with empty type p5 self i j pc' iinv5[of i j]
    show ?thesis
     unfolding IInv_def TypeOK_def p5_def After_def
     apply (cases "self = j")
      using unreadj apply force
      apply clarsimp
     done
   qed
  qed
  from 1 2 3 4 5 show ?thesis
   unfolding IInv_def by blast
next
  assume empty: "unread[self] ≠ {}"
  from p5 type iinvi self i empty
  have 1: "(num'[i] = 0) = (pc'[i] ∈ {"p1", "p2", "p3"})"
   unfolding TypeOK_def IInv_def p5_def
   by(cases "self = i", clarsimp+)
  from p5 type iinvi self i empty
  have 2: "flag'[i] = (pc'[i] ∈ {"p2", "p3", "p4"})"
   unfolding TypeOK_def IInv_def p5_def
   by(cases "self = i", clarsimp+)
  have 3: "pc'[i] ∈ {"p5", "p6"} ⇒
   (∀j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
  proof (rule+)
   fix j
   assume pc': "pc'[i] ∈ {"p5","p6"}" and j: "j ∈ (ProcSet \ unread'[i]) \ {i}"
   with empty type p5 self i j iinv3[of i j]
   show "After(j,i,unread',max',flag',pc',num',nxt')"
    unfolding TypeOK_def After_def GG_def p5_def
    apply(cases "self = j", force, clarsimp)
    by(cases "self = i", simp+)
  qed
  have 4: "pc'[i] = "p6"
```

```
  ∧ (  (pc'[nxt'[i]] = "p2") ∧ i ∉ unread'[nxt'[i]]
  ∨ (pc'[nxt'[i]] = "p3"))
  ⇒ max'[nxt'[i]] ≥ num'[i]"
proof -
  from empty p5
  obtain k where p5:
    "pc[self] = "p5""
    "k ∈ unread[self]" "nxt' = [nxt EXCEPT ![self] = k]"
    "¬ flag[nxt'[self]] ∧ pc' = [pc EXCEPT ![self] = "p6"]"
    "num' = num" "flag' = flag" "unread' = unread" "max' = max"
    by(auto simp: p5_def)
  with type self have kproc: "k ∈ ProcSet"
    by (auto simp: TypeOK_def)
  with iinv have iinvk: "IInv(k, unread, max, flag, pc, num, nxt)" ..
  show ?thesis
  proof (cases "self = i")
    case True
    with empty p5 type iinvi self i iinvk
    show ?thesis
      unfolding TypeOK_def IInv_def
      by(cases "self = k", clarsimp, force simp: kproc)
  next
    case False
    with empty p5 type iinvi self i
    show ?thesis
      unfolding TypeOK_def IInv_def
      apply(cases "self = nxt[i]")
        apply clarsimp
        using nxti apply clarsimp
      done
  qed
qed
have 5: "(pc'[i] ∈ {"p7", "p8"}) ⇒ (∀ j ∈ P \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
proof (rule+)
  fix j
  assume pc': "pc'[i] ∈ {"p7","p8"}" and j: "j ∈ ProcSet \ {i}"
  from j type have mx: "pc[j] = "p2" ⇒ max[j] ∈ Nat"
    by (auto simp: TypeOK_def)
  from j iinv have iinvj: "IInv(j, unread, max, flag, pc, num, nxt)" by auto
  from empty type p5 self i j pc' iinv5[of i j]
  show "After(j,i,unread',max',flag',pc',num',nxt')"
    apply (cases "self = j")
      apply (clarsimp simp: TypeOK_def p5_def After_def)
      apply (cases "self = i")
      apply (clarsimp simp: TypeOK_def p5_def After_def)+
      done
qed
from 1 2 3 4 5 show ?thesis
```

```
    unfolding IInv_def by blast
  qed
next
  assume p6: "p6(self, unread, max, flag, pc, num, nxt, unread', max', flag', pc', num', nxt')"
  from p6 iinvi type i
  have 1: "(num'[i] = 0) = (pc'[i] ∈ {"p1", "p2", "p3"})"
   and 2: "flag'[i] = (pc'[i] ∈ {"p2", "p3", "p4"})"
   unfolding TypeOK_def IInv_def p6_def
   by (cases "self = i", clarsimp, clarsimp)+
  have 3: "pc'[i] ∈ {"p5", "p6"} ⇒
   (∀ j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
  proof(rule+)
   fix j
   assume pc': "pc'[i] ∈ {"p5", "p6"}" and j: "j ∈ ProcSet \ unread'[i] \ {i}"
   from iinv j
   have iinvj: "IInv(j, unread, max, flag, pc, num, nxt)" by auto
   from type j
   have nxtj: "pc[j] = "p6" ⇒ nxt[j] ∈ ProcSet ∧ nxt[j] ≠ j"
    by (auto simp: TypeOK_def)
   show "After(j, i, unread', max', flag', pc', num', nxt')"
   proof (cases "self = i")
    assume selfi: "self = i"
    show ?thesis
    proof (cases "j ∈ unread[self]")
     assume junread: "j ∈ unread[self]"
     show ?thesis
     proof(cases "num[j] = 0")
      case True
      with j selfi i pc' p6 type self iinvi junread iinvj
      show ?thesis
       by (clarsimp simp: p6_def TypeOK_def IInv_def After_def nat_gt0_not0 condElse, force)
     next
      assume numj: "num[j] ≠ 0"
      show ?thesis
      proof (cases "pc[j] ∈ {"p4", "p5", "p6"}")
       assume pcj: "pc[j] ∈ {"p4", "p5", "p6"}"
       show ?thesis
       proof (cases "i ∈ unread[j]")
        case True
        with j selfi i pc' p6 type self iinvi junread numj pcj
        show ?thesis
         by (clarsimp simp: p6_def TypeOK_def IInv_def After_def GG_def nat_gt0_not0)
       next
        assume notunread: "i ∉ unread[j]"
        show ?thesis
        proof (cases "j < i")
         case True
         from True j selfi i pc' p6 type self iinvi junread numj pcj iinv3[of j i] nxti notunread
```

```
            show ?thesis
              using procInNat nat_less_antisym_false[of j i]
               by (clarsimp simp: p6_def TypeOK_def IInv_def After_def GG_def nat_gt0_not0 condElse)
(force simp: nat_less_antisym_leq_false)
          next
            case False
            with j selfi i pc' p6 type self iinvi junread numj pcj iinv3[of j i] nxti notunread
            show ?thesis
              unfolding p6_def TypeOK_def IInv_def After_def GG_def
              proof (clarsimp simp: nat_gt0_not0 condElse nat_not_less procInNat
                        dest!: leq_neq_trans'[of "i" "nxt[i]"])
                assume
                  "pc[nxt[i]] = "p5" ∨ pc[nxt[i]] = "p6" ⟹ num[nxt[i]] < num[i]"
                  "num[i] ≤ num[nxt[i]]" "nxt[i] ∈ ProcSet" "i ∈ ProcSet"
                  "num ∈ [ProcSet → Nat]"
                thus "pc[nxt[i]] ≠ "p5" ∧ pc[nxt[i]] ≠ "p6""
                  by (auto simp: nat_less_antisym_leq_false)
              qed
            qed
          qed
        next
          assume pcj: "pc[j] ∉ {"p4", "p5", "p6"}"
          show ?thesis
          proof (cases "pc[j] ∈ {"p7", "p8"}")
            case True
            assume pcj: "pc[j] ∈ {"p7", "p8"}"
            show ?thesis
            proof (cases "j < i")
              case True
              with j selfi i pc' p6 type self iinvi junread numj pcj iinvj nxti iinv5[of "nxt[i]" i]
              show ?thesis
                unfolding p6_def TypeOK_def IInv_def After_def GG_def
                using nat_less_antisym_false[of j i]
                apply (clarsimp simp: nat_gt0_not0 procInNat condElse)
                by (clarsimp simp: nat_less_antisym_leq_false)
            next
              case False
              with j selfi i pc' p6 type self iinvi junread numj pcj iinv5[of j i] nxti
              show ?thesis
                unfolding p6_def TypeOK_def IInv_def After_def GG_def
                apply (clarsimp simp: nat_gt0_not0 condElse nat_not_less procInNat condElse)
                by (clarsimp dest!: leq_neq_trans' simp: nat_less_antisym_leq_false)
            qed
          next
            assume pcj2: "pc[j] ∉ {"p7", "p8"}"
            with j selfi i pc' p6 type self iinvi junread numj pcj iinvj nxti
            show ?thesis
              apply (clarsimp simp: p6_def TypeOK_def IInv_def After_def GG_def nat_gt0_not0)
```

```
            by (erule funcSetE[where f="pc" and x="nxt[i]"], simp+)
          qed
        qed
      qed
    next
      assume "j ∉ unread[self]"
      with j selfi i pc' p6 type self iinv3[of i j]
      show ?thesis
        by (clarsimp simp: p6_def TypeOK_def After_def condElse, auto)
    qed
  next
    assume selfi: "self ≠ i"
    show ?thesis
    proof (cases "self = j")
      assume selfj: "self = j"
      show ?thesis
      proof (cases "j < nxt[j]")
        case True
        with j pc' selfi selfj i p6 type self iinv3[of i j]
        show ?thesis
          apply (clarsimp simp: p6_def TypeOK_def After_def GG_def nat_gt0_not0)
          by (auto dest: nat_less_trans nat_less_leq_trans)

      next
        case False
        with j pc' selfi selfj i p6 type self iinv3[of i j]
        show ?thesis
          using nxtj nat_not_less procInNat
        by (clarsimp simp: p6_def TypeOK_def After_def GG_def nat_gt0_not0 condElse leq_neq_iff_less)
(clarsimp simp: nat_less_antisym_leq_false)
      qed
    next
      case False
      with j selfi i pc' p6 type self iinv3[of i j]
      show ?thesis
        by (clarsimp simp: p6_def TypeOK_def After_def)
    qed
  qed
qed
have 4: "pc'[i] = "p6"
  ∧ (  (pc'[nxt'[i]] = "p2") ∧ i ∉ unread'[nxt'[i]]
  ∨ (pc'[nxt'[i]] = "p3"))
  ⇒ max'[nxt'[i]] ≥ num'[i]"
proof (cases "self = nxt[i]")
  case True
  with p6 type self i show ?thesis
    by (clarsimp simp: p6_def TypeOK_def)
next
```

```
    case False
    with p6 type iinvi self i show ?thesis
     apply (clarsimp simp: TypeOK_def IInv_def p6_def)
     by (cases "self = i", simp, simp add: nxti)
   qed
   have 5: "(pc'[i] ∈ {"p7", "p8"}) ⇒ (∀ j ∈ P \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
   proof (rule+)
    fix j
    assume pc': "pc'[i] ∈ {"p7","p8"}" and j: "j ∈ ProcSet \ {i}"
    show "After(j,i,unread',max',flag',pc',num',nxt')"
    proof (cases "self = j")
     case True
     assume selfj: "self = j"
     show ?thesis
     proof (cases "j < i")
      case True
      with type p6 i pc' j self selfj iinv5[of i j]
      show ?thesis
       using nat_less_antisym_false[of j i] procInNat
       apply (clarsimp simp: p6_def TypeOK_def IInv_def After_def GG_def nat_gt0_not0)
       by (clarsimp simp: nat_less_antisym_leq_false)
     next
      case False
      with type p6 i pc' j self selfj iinv5[of i j]
      show ?thesis
       using nat_not_less procInNat
       apply (clarsimp simp: p6_def TypeOK_def IInv_def After_def GG_def nat_gt0_not0 condElse)
       by (clarsimp dest!: leq_neq_trans' simp add: nat_less_antisym_leq_false)
     qed
    next
     case False
     with type iinv5[of i j] p6 i pc' j show ?thesis
      apply (clarsimp simp: TypeOK_def IInv_def After_def p6_def)
      by (cases "self = i", simp, simp)
    qed
   qed
   from 1 2 3 4 5 show ?thesis unfolding IInv_def by blast
  next
   assume p7: "p7(self, unread, max, flag, pc, num, nxt, unread', max', flag', pc', num', nxt')"
   from p7 type iinvi self i
   have 1: "(num'[i] = 0) = (pc'[i] ∈ {"p1", "p2", "p3"})"
    and 2: "flag'[i] = (pc'[i] ∈ {"p2", "p3", "p4"})"
    unfolding TypeOK_def IInv_def p7_def
    by (cases "self = i", clarsimp, clarsimp)+
   have 3: "pc'[i] ∈ {"p5", "p6"} ⇒
    (∀ j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
   proof (rule+)
    fix j
```

```
  assume pc': "pc'[i] ∈ {"p5","p6"}" and j: "j ∈ (ProcSet \ unread'[i]) \ {i}"
  show "After(j,i,unread',max',flag',pc',num',nxt')"
  proof (cases "self = i")
    case True
    with type p7 iinv3[of i j] j i pc'
    show ?thesis
      by (clarsimp simp: TypeOK_def p7_def)
  next
    assume selfi: "self ≠ i"
    with type p7 iinv3[of i j] j i pc' selfi
    show ?thesis
      unfolding TypeOK_def p7_def After_def
      by (cases "self = j", clarsimp, force)
  qed
qed
have 4: "pc'[i] = "p6"
       ∧ (  (pc'[nxt'[i]] = "p2") ∧ i ∉ unread'[nxt'[i]]
          ∨ (pc'[nxt'[i]] = "p3"))
       ⇒ max'[nxt'[i]] ≥ num'[i]"
proof (cases "self = i")
  case True
  with p7 type self i show ?thesis
    by (clarsimp simp: TypeOK_def p7_def)
next
  case False
  with p7 type iinvi self i show ?thesis
    apply (clarsimp simp: TypeOK_def IInv_def p7_def)
    by (cases "self = nxt[i]", simp, simp add: nxti)
qed
have 5: "(pc'[i] ∈ {"p7", "p8"}) ⇒ (∀ j ∈ P \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
proof (rule+)
  fix j
  assume pc': "pc'[i] ∈ {"p7","p8"}" and j: "j ∈ ProcSet \ {i}"
  show "After(j, i, unread', max', flag', pc', num', nxt')"
  proof (cases "self = j")
    case True
    with p7 type iinv5[of i j] self i pc' j show ?thesis
      by (clarsimp simp: TypeOK_def p7_def After_def)
  next
    case False
    with p7 type iinv5[of i j] self i pc' j show ?thesis
      apply (clarsimp simp: TypeOK_def p7_def After_def nat_gt0_not0)
      by (cases "self = i", simp, simp)
  qed
qed
from 1 2 3 4 5 show ?thesis
  unfolding IInv_def by blast
next
```

```
assume p8: "p8(self, unread, max, flag, pc, num, nxt, unread', max', flag', pc', num', nxt')"
have 1: "(num'[i] = 0) = (pc'[i] ∈ {"p1", "p2", "p3"})"
 and 2: "flag'[i] = (pc'[i] ∈ {"p2", "p3", "p4"})"
 using p8 type iinvi self i
 unfolding p8_def TypeOK_def IInv_def
 by (cases "self = i", clarsimp, clarsimp)+
have 3: "pc'[i] ∈ {"p5", "p6"} ⇒
(∀j ∈ (P \ unread'[i]) \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
proof (rule+)
 fix j
 assume pc': "pc'[i] ∈ {"p5","p6"}" and j: "j ∈ (ProcSet \ unread'[i]) \ {i}"
 show "After(j,i,unread',max',flag',pc',num',nxt')"
 proof (cases "self = i")
  case True
  with type p8 j i pc'
  show ?thesis
   by (clarsimp simp: TypeOK_def p8_def)
 next
  assume selfi: "self ≠ i"
  with type p8 iinv3[of i j] j i pc' selfi
  show ?thesis
   unfolding TypeOK_def p8_def After_def GG_def
   by (cases "self = j", clarsimp+)
 qed
qed
have 4: "pc'[i] = "p6"
      ∧ ( (pc'[nxt'[i]] = "p2") ∧ i ∉ unread'[nxt'[i]]
        ∨ (pc'[nxt'[i]] = "p3"))
      ⇒ max'[nxt'[i]] ≥ num'[i]"
proof (cases "self = i")
 case True
 with p8 type self i show ?thesis
  by (clarsimp simp: p8_def TypeOK_def)
next
 case False
 with p8 type iinvi self i show ?thesis
  apply (clarsimp simp: p8_def TypeOK_def IInv_def)
  by (cases "self = nxt[i]", simp, clarsimp simp: nxti)
qed
have 5: "(pc'[i] ∈ {"p7", "p8"}) ⇒ (∀j ∈ P \ {i} : After(j,i,unread',max',flag',pc',num',nxt'))"
proof (rule+)
 fix j
 assume pc': "pc'[i] ∈ {"p7","p8"}" and j: "j ∈ ProcSet \ {i}"
 show "After(j, i, unread', max', flag', pc', num', nxt')"
 proof (cases "self = i")
  case True with type self i p8 pc' j show ?thesis
   by (clarsimp simp: TypeOK_def p8_def)
 next
```

```
        case False with type self i p8 pc' j iinv5[of i j] show ?thesis
          apply (clarsimp simp: TypeOK_def p8_def After_def GG_def)
          by (cases "self = j", simp, force)
      qed
    qed
    from 1 2 3 4 5 show ?thesis unfolding IInv_def by blast
  qed
 qed
qed

end
```