Título

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Glauber Módolo Cabral e aprovada pela Banca Examinadora.

Campinas, 12 de junho de 2009.

Arnaldo Vieira Moura (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Substitua pela ficha catalográfica

(Esta página deve ser o verso da página anterior mesmo no caso em que não se imprime frente e verso, i.é., até 100 páginas.)

Substitua pela folha com as assinaturas da banca

Instituto de Computação

Universidade Estadual de Campinas

Título

Glauber Módolo Cabral[1]

Abril de 2009

Banca Examinadora:

- Arnaldo Vieira Moura (Orientador)
- Charlie Brown

  Departamento de Apicultura Subterrânea Computacional
- Donald Duck
- Era Stoteles
- Fairy Tail

[1]Suporte financeiro do CNPq (processo 132039/2007-9)

Resumo

O resumo deve conter no máximo 500 palavras...

Abstract

The abstract must contain at most 500 words...

Agradecimentos

Eu gostaria de agradecer a...

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Capítulo 6 revisão de conceitos: Lógica de Primeira Ordem; Lógica de Segunda Ordem; Laziness.

Capítulo 2 revisão de linguagens relacionadas. Situar cada linguagem, com pequena descrição. Explicar onde CASL/HasCASL se encaixam e onde meu trabalho entra.

Capítulo 3 tutorial de HasCASL.

Capítulo 4 explicação passo a passo da criação da biblioteca.

Capítulo 5 explicação do uso de Hets + Isabelle para a contrução das provas. Listar as provas ou deixar no anexo?

Capítulo 8 conclusões e trabalhos futuros.

Capítulo B lista de provas. Deveria estar em Capítulo 5?

Capítulo 2

Revisão Bibliográfica de Linguagens de Especificação Formal Correlatas

2.1 Extended ML

Extended ML (EML) [3] consiste em uma linguagem para desenvolvimento formal de sistemas corretos com relação à uma especificação. EML foi construída a partir da linguagem de programação funcional *Standard ML (SML)*, estendendo a sintaxe e a semântica desta última para alcançar o formalismo necessário para a execução de inferências lógicas.

A linguagem de programação funcional *Standard ML (SML)* é composta por duas sublinguagens. A primeira, nomeada linguagem *core*, provê a definição de tipos e valores (variáveis, funções e elementos) daqueles tipos. A segunda, chamada linguagem *module*, permite a definição e a combinação de unidades autocontidas de programas codificadas na primeira linguagem.

A linguagem *core* é fortemente tipificada com um sistema de tipos que agrega tipos polimórficos, união disjunta de tipos, produtos de tipos, tipos que são funções de ordens superiores, tipos recursivos e a possibilidades de definição de tipos abstratos e concretos pelo usuário. A semântica da linguagem também ~~possui~~ reflete características de linguagens imperativas de programação, a saber: tratamento de exceções e referências (ponteiros) tipificadas. O tratamento de entrada e saída de dados é realizado através de *streams*, que associam o fluxo de entrada com produtores (teclado, por exemplo) e o fluxo de saída com consumidores (monitor).

A linguagem *module* fornece mecanismos para a modularização de programas. Uma assinatura (*signature*) fornecem uma interface que é implementada por uma estrutura (*structure*). Uma assinatura define tipos, valores, exceções e subestruturas que devem ser implementados pelas estruturas posteriormente associadas a esta assinatura. A linguagem ainda apresenta functores, equivalentes a estruturas parametrizadas, que são aplicados sobre outras estruturas a fim de se obter novas estruturas. Um functor possui uma assinatura de entrada, que descreve as estruturas aos quais pode ser aplicado, e uma assinatura opcional de saída, que define a estrutura resultante da aplicação do mesmo sobre outras estruturas. Estruturas e functores definem módulos e as assinaturas, por sua vez, impõe~~s~~m restrições às definições de módulos, além de definirem quais declarações dos módulos serão visíveis pelos usuários destes módulos.

Além de ter sua sintaxe e sua semântica formalmente definidas, a linguagem SML possui uma biblioteca padrão definida e que deve ser implementada ~~pelos~~ por compiladores [2]. A biblioteca especifica interfaces e operações para tipos básicos de dados, como inteiros, caracteres e cadeias de caracteres, listas, tipos opcionais e vetores mutáveis e imutáveis. Também são definidos o suporte para operações de entrada e saída binária e textual, além de interfaces para serviços do sistema operacional e para comunicação em rede.

A linguagem EML permite o desenvolvimento formal de software através de passos individualmente verificados. Ela engloba todos os estágios deste desenvolvimento, desde a especificação de alto nível até o programa final. Por ter como produto final um programa modular escrito em SML, um grande subconjunto desta última linguagem forma uma sublinguagens executável de EML.

Além das características pertencentes às linguagens de programação funcional, EML possui outras duas características particularmente interessantes para a aplicação em métodos formais. Primeiramente, permite total modularização para a ~~construção~~ modelagem de sistemas complexos. A modularização é um pré-requisito para a aplicação de métodos formais ~~em~~ no desenvolvimento de exemplos ~~de~~ complexi~~dades reais~~os, e EML foi projetada visando a ser empregada neste contexto. Em segundo, a sintaxe e a semântica da linguagem são formalmente definidas. ? Com a definição formal, torna-se possível fazer inferências por meio de lógica formal (to reason formally about) sobre o comportamento de programas escritos em SML, o que permite, por sua vez, o desenvolvimento de provas de corretude segundo uma dada especificação formal.

EML foi desenvolvida com o propósito de ser uma extensão mínima à linguagem SML. Tem ~~tendo~~ como princípio o desenvolvimento de uma linguagem para especificar sistemas modulares de *software* escritos em SML ao invés do desenvolvimento de uma linguagem de especificação de propósito geral. Devido a essa abordagem, EML pode ser facilmente aprendida por aqueles que possuem contato anterior com SML. No entanto, devido à complexidade da linguagem SML, a sintaxe de EML também é complexa e seu manuseio no processo de especificação e verificação torna-se difícil. Uma outra desvantagem da linguagem é a falta de suporte a ferramentas automáticas de verificação, tais como provadores de teoremas. [4]

A principal extensão de EML com relação à sintaxe de SML é a inclusão de axiomas nas assinaturas e no corpo dos módulos. Os axiomas definem propriedades que devem ser satisfeitas por todas as estruturas que implementarem a interface onde eles foram definidos. Assim como os tipos dos valores definidos nas interfaces de SML permitem com que cada módulo seja compilado separadamente, a inclusão de axiomas nas interfaces de EML permite com que propriedades dos módulos descritos por uma interface possam ser verificadas sem que seja necessário verificar o conteúdo do módulo. Isto ~~permite~~ cria a facilidade para que implementações diferentes de uma mesma interface sejam criadas e possam ser utilizadas sem afetar a corretude das provas anteriormente desenvolvidas.

confuso. Re-escrever.

Ao contrário de SML, as assinaturas são opcionais para algumas estrutudas, todas as estruturas definidas em EML devem possuir, obrigatoriamente, uma assinatura. No caso dos functores, ambas as assinaturas de entrada e saída do mesmo se tornam obrigatórias. Além do casamento entre nomes e tipos entre as entidades de uma assinatura e de um módulo, o corpo do módulo deve estar correto, ou seja, deve satisfazer todos os axiomas da assinatura a que está associado.

Uma terceira estensão permite que o desenvolvimento de um programa em EML comece com a definição de módulos de forma abstrata. Pode-se utilizar um sinal de interrogação no lugar da definição do corpo de um módulo, permitindo que sua definição seja incluída em refinamentos posteriores. Axiomas definindo o comportamento esperado da estrutura com declaração abstrata são declarados logo em seguida. Nos refinamentos posteriores, o sinal de interrogação é substituído por código escrito em SML, o qual deve obedecer às propriedades definidas pelos axiomas. O desenvolvimento em EML é encerrado quando todos os sinais de interrogação são substituídos por código SML executável e os axiomas transformados em documentação.

EML possui um analisador sintático para verificar especificações e o código SML resultante do processo de especificação pode ser analisado e ~~compilado~~ gerado pelo compilador da linguagem SML.

O exemplo a seguir define uma estrutura representando uma pilha de inteiros. Para tal, define duas assinaturas, um functor e a estrutura em si. A primeira assinatura define um tipo para representar elementos presentes dentro da pilha. A outra assinatura descreve o conteúdo de uma pilha. Segundo a assinatura, uma pilha possui uma estrutura, representando os elementos da pilha; o tipo associado aos elementos que obedecem essa assinatura; quatro valores, representando a pilha vazia e três operações sobre pilhas; e dois axiomas, definindo propriedades que as operações devem obedecer. O functor é uma estrutura de criação de pilhas, ou seja, ao ser aplicado sobre um elemento que obedeça a assinatura `OBJ`, deverá retornar uma estrutura que obedeça a assinatura `STACK` e onde a estrutura `Obj` da assinatura anterior será o próprio elemento sobre o qual o functor foi aplicado. Por fim, cria-se uma estrutura `IntStack`, que obedece à assinatura `STACK`, através da aplicação do functor `Stack` sobre uma estrutura que obedece à assinatura `OBJ` e cujo tipo `object` é associado ao tipo primitivo `int`.

```
signature OBJ =
  sig
    type object
  end
signature STACK =
  sig
    structure Obj : OBJ
    type stack
    val empty : stack
    val push : Obj.object * stack -> stack
    val pop : stack -> stack
    val top : stack -> Obj.object
    axiom pop(push(a,s)) = s
    axiom top(push(a,s)) = a
  end
functor Stack(O : OBJ) :  sig include STACK
```

```
                              sharing Obj = O
                          end = ?
structure IntStack : STACK = Stack(struct
                                        type object = int
                                   end )
```

Capítulo 3

A Linguagem de Especificação Algébrica HasCASL

Neste capítulo, apresenta-se um tutorial introdutório à linguagem de especificação algébrica HasCasl. O material aqui apresentado teve como base o manual da linguagem Casl [1] e o documento que descreve a linguagem HasCasl [5]. Para uma referência completa, os documentos citados devem ser consultados.

3.1 Introdução

A linguagem de especificação algébrica *Common Algebraic Specification Language (*Casl*)* foi concebida para ser a linguagem padrão na área de especificação algébrica. As características da linguagem foram extraídas de outras linguagens de forma a obter uma linguagem que pudesse abranger a maioria das funcionalidades presentes nas demais linguagens de especificação algébrica. Como uma só linguagem não conseguiria captar todas as funcionalidades existentes, Casl foi projetada para permitir extensões e sublinguagens, permitindo, assim, que características que exijam outros paradigmas ou a ausência de determinadas características pudessem ser incorporadas. As sublinguagens são criadas por restrições semânticas ou sintáticas à linguagem Casl e suas extensões servem para implementar diferentes paradigmas de programação.

A linguagem HasCasl é uma extensão à linguagem Casl para incluir suporte à logica de segunda ordem. Seu núcleo consiste em uma lógica de segunda ordem de funções parciais construídas sobre o λ-cálculo de Moggi. Este núcleo é estendido com subtipos e polimorfismo baseado em classes de tipos, incluindo-se construtores de tipos de segunda ordem e construtores de classes. Devido a vários atalhos sintáticos (*syntax sugar*), existe um subconjunto da linguagem que pode ser executado e se assemelha intimamente com a linguagem de programação Haskell.

Os elementos sintáticos da linguagem Casl estão presentes em todas as suas extensões, dentre elas, a linguagem HasCasl. Algumas características semânticas possuem sintaxe diferente, mas equivalente, nas linguagens Casl e HasCasl. Alguns elementos, ainda, possuem a mesma finalidade embora possuam diferenças semânticas importantes nas duas linguagens.

O presente tutorial concentra-se em apresentar a linguagem HasCasl, citando, quando existentes, as equivalências e diferenças com elementos da sintaxe de Casl, além de introduzir a sintaxe de Casl que deve ser utilizada pelas suas extensões. As características da linguagem Casl, quando não informado o contrário, continuam válidas para a linguagem HasCasl.

3.2 Tipos de especificações em Casl

As especificações escritas em Casl podem ser de quatro tipos:

- Especificações Básicas: contêm declarações – de tipos e de funções –, axiomas – definindo e relacionando operações – e propriedades – definidas através de teoremas e que restringem o comportamento de tipos e funções;

- Especificações Estruturais: permitem que as especificações básicas sejam combinadas para formar especificações mais complexas;
- Especificações Arquiteturais: definem como devem ser separadas as várias especificações na implementação, de forma que seja possível o reuso de especificações dependentes entre si;
- Especificações de Bibliotecas: definem conjuntos de especificações, com funcionalidades que permitem controle de versão e de bibliotecas distribuídas pela Internet.

Como este trabalho não contempla implementação de especificações, as especificações arquiteturais não foram abordadas. As demais especificações são detalhadas em seções individuais a seguir.

É importante salientar que a semântica das especificações básicas e estruturais é independente da instituição empregada, ou seja, independe da lógica empregada para as especificações básicas. Desta forma, para definir extensões da linguagem CASL basta definir a lógica da nova linguagem na forma de uma estrutura categórica conhecida como instituição. Isto significa definir as noções de assinatura, modelo, sentença e satisfação (de predicados).

3.3 Modelos de interpretação de especificações escritas em CASL

A linguagem CASL permite três modelos de interpretação semântica para as especificações, a saber: modelo *loose*, modelo *generated* e modelo *free*. O modelo padrão utilizado é o modelo *loose*. Para indicar os outros dois modelos de interpretação, CASL utiliza, respectivamente, os modificadores de tipos `generated` e `free`

O modelo *loose* permite que os modelos de uma especificação abranjam todos aqueles modelos nos quais as funções declaradas possuam as propriedades especificadas, sem fazer restrições aos conjuntos de valores do domínio das funções. Já o modelo *generated* exige que todos os possíveis elementos dos tipos pertencentes ao domínio das funções sejam formados apenas pelos construtores dos respetivos tipos, proibindo a existência de elementos inatingíveis no domínio. Por sua vez, o modelo *free* requer que os valores dos elementos dos tipos do domínio sejam diferentes entre si, exceto se a igualdade dos mesmos for expressa por axiomas. Isto impede a coincidência acidental entre elementos dos tipos do domínio.

Embora os três modelos presentes em CASL estejam presentes em todas as suas extensões ou sublinguagens, os tipos de dados expressos em HASCASL são frequentemente modelados por teorias do modelo *free*. Um tipo de dado declarado neste modelo possui a propriedade *no junk, no confusion*, ou seja, só há elementos do tipo em questão formados pelos construtores declarados e todos os elementos são diferentes entre si. Este modelo provê a semântica inicial de interpretação de especificações e, dessa forma, evita a necessidade de se criar axiomas que neguem a igualdade entre construtores de tipo diferentes.

Predicados definidos em modelos *free* são verdadeiros apenas se decorrem dos axiomas declarados na especificação, sendo falsos nos demais casos. Como decorrência deste comportamento, apenas os casos para os quais o predicado deve ser verdadeiro precisam ser axiomatizados; os demais casos serão considerados falsos automaticamente.

Nos modelos *free*, pode-se definir predicados e funções por indução nos construtores dos tipos de dados de forma segura. Uma boa prática para defini-los é axiomatizar as funções para cada um dos construtores do(s) tipo(s) do(s) parâmetro(s). Este processo é conhecido como *case distinction*.

3.4 Especificações Básicas

A semântica de uma especificação básica possui dois elementos:

- uma assinatura composta pelos símbolos introduzidos pela especificação; e
- uma classe de modelos correspondendo às interpretações da assinatura que satisfazem os axiomas e as restrições da especificação.

A assinatura contém as definições dos símbolos de tipos e de funções, dos axiomas e das propriedades. Geralmente, os símbolos são declarados e restringidos em uma mesma declaração, embora seja possível declarar um símbolo e restringir o seu comportamento posteriormente.

Por serem as especificações mais comuns, serão tratadas no texto apenas por especificações; os demais tipos de especificações serão explicitados, quando não puderem ser distinguidos pelo contexto.

3.4.1 Casl

Casl possui sintaxe específica para declaração de tipos (*sorts*), subtipos (*subsorts*), operações e predicados. Em Casl, um tipo (*sort*) é interpretado como um conjunto cujos elementos são representações abstratas de dados processados por programas. Este conjunto é chamado *carrier set*. Dessa forma, um tipo (*sort*) equivale a um tipo em uma linguagem de programação. Pode-se declarar tipos simples, como inteiros (`Int`) e listas (`List`); e tipos compostos, como listas de inteiros (`List [Int]`).

A relação de subtipos (*subsorts*) é interpretada como uma função injetiva que mapeia cada um dos elementos do subtipo para um único elemento do supertipo e recebe o nome de *embedding*. Por exemplo, cada número natural pode ser mapeado para o inteiro positivo correspondente, no caso dos tipos `Nat` e `Int`, e um caractere pode ser mapeado para uma cadeira de caracteres formada apenas pelo caractere mapeado, no caso dos tipos `Char` e `String`.

Uma operação é formada por um nome e por um perfil, o qual indica o número e os tipos (*sorts*) dos argumentos e o tipo (*sort*) do resultado da operação. Operações podem ser totais – definidas para todos os elementos dos tipos (*sorts*) – ou parciais – definidas para um subconjuntos de elementos dos tipos (*sorts*). A aplicação de uma operação sobre um parâmetro indefinido sempre resulta em um valor indefinido, independente da função ser total ou parcial. Quando não há parâmetros, a operação é considerada uma constante e representa um elemento do tipo (*sort*) a que pertence.

Um predicado, assim como uma operação, consiste de um nome e um perfil, sendo que este último não possui um tipo (*sort*) para resultado. Predicados são interpretados como uma relação sobre o produto cartesiano entre os *carrier sets* dos tipos (*sorts*) dos parâmetros do predicado e são utilizados para formar fórmulas atômicas ao invés de termos. Um predicado é verdadeiro quando a tupla formada pelos seus parâmetros está contida na relação que define o predicado.

Diferentemente das funções, quando o valor de algum dos parâmetros de um predicado é indefinido, o predicado é falso. Desta forma, a lógica continua apresentando apenas dois valores, *verdadeiro* e *falso*. Em contraste, uma operação booleana poderia apresentar três valores possíveis, já que resultaria em um valor indefinido quando algum de seus parâmetros fosse indefinido. Uma outra diferença entre predicados e operações diz respeito ao conceito de modelo inicial. Predicados com lógicas de dois valores podem ser representados por operações parciais com um tipo (*sort*) resultante formado por apenas um elemento, sendo que o predicado é verdadeiro sempre que estiver definido para os parâmetros recebidos.

Operações e predicados podem ser sobrecarregados, ou seja, um mesmo nome pode possuir diferentes perfis associados. No entanto, a sobrecarga precisa ser compatível no que diz respeito ao *embedding* entre subtipos. Isto significa que, dados os tipos (*sorts*) `A` e `B`, com `A` sendo subtipo de `B`, uma operação `operação` e um predicado `predicado`, definidos sobre ambos os tipos (*sorts*), a interpretação de `operação` deve ser tal que não faça diferença aplicar a função de *embedding* nos argumentos de `operação` ou no resultado de `operação` e a interpretação de `predicado` deve ser tal que não faça diferença aplicar ou não a função de *embedding* aos argumentos de `predicado`.

Os axiomas em CASL são formulas de lógica de primeira ordem, com as interpretações padrões para os quantificadores e conectivos lógicos. As variáveis das fórmulas representam qualquer elemento dos conjuntos *carrier sets* dos tipos (*sorts*) especificados. Os axiomas são introduzidos por um ponto final antes de sua declaração.

Certas fórmulas denotam propriedades, que devem decorrer de outros axiomas. Estas fórmulas são consideradas teoremas e devem ser anotadas com `% implied` a fim de indicar a necessidade de prova da fórmula. Tal necessidade de prova irá gerar um teorema a ser verificado pela ferramenta de verificação de teoremas escolhida.

Além da aplicação usual de predicados, fórmulas escritas em CASL podem representar equações (universais ou existenciais), assertivas de definição e assertivas de pertinência em tipos (*sorts*). Uma equação existencial é verdadeira quando os valores dos seus termos são definidos e iguais. Já uma equação universal é verdadeira também quando ambos os seus termos são indefinidos. Assim como nos predicados, as equações e assertivas de definição e de pertinência em tipos (*sorts*) são falsas quando um de seus termos são indefinidos, mantendo apenas dois valores na lógica.

3.4.2 HasCasl

Em HasCasl, a sintaxe permite declarar tipos (*types*), subtipos (*subtypes*) e funções. Os predicados são considerados um caso particular de funções. Por questões de compatibilidade, HasCasl trata tipos (*sorts*) e tipos (*types*) como equivalentes. Já operações e funções diferem apenas quanto ao comportamento em relação a subtipos. A visibilidade das declarações na linguagem é linear, ou seja, os elementos precisam ser definidos antes de serem utilizados.

Os tipos são construídos a partir de tipos básicos declarados através da palavra-chave `type` e, por padrão, como em Casl, os tipos são interpretados pela semântica *loose*. A partir dos tipos básicos e do tipo unitário `Unit`, os tipos são gerados de forma indutiva através do produto entre tipos ($t_1 * t_2 * \ldots * t_n$) e de funções com tipos parciais ($s \rightarrow ?\ t$) e totais ($s \rightarrow\ t$).

Pode-se abreviar um tipo através de um *sinônimo*, criado com a palavra-chave `type`, e o tipo a que o sinônimo se refere é chamado uma expansão. Embora a mesma palavra-chave seja utilizada, sinônimos não são tipos básicos. Eles podem ser definidos apenas uma vez e definições recursivas não são permitidas. A título de exemplo, a seguir são definidos dois tipos básicos – `S` e `T` – e um sinônimo – `LongType`. O tipo `WrongLongType`, recursivamente definido, é um exemplo de tipo inválido e aparece precedido do caractere de comentário de linha nas declarações a seguir:

```
type S,T
type LongType := (S * S * S) -> T
%% type WrongLongType := LongType * T -> S
```

Termos são formados por variáveis ou operações e, internamente, são sempre anotados com o tipo a que pertencem. Um termo `t` do tipo `T` com o seu tipo anotado explicitamente resulta no termo `t: T`. Quando um termo não possui uma anotação de tipo, o tipo do mesmo é inferido pelo contexto, que compreende todas as definições de variáveis e funções alcançáveis no ponto em que o termo aparece. Desta forma, só é necessário anotar o tipo de um termo explicitamente quando o mesmo não puder ser inferido sem ambiguidade pelo contexto.

A declaração de variáveis pode ser local ou global e é sempre universalmente quantificada. Variáveis globais são introduzidas pela palavra-chave `var`. Esta declaração inicia uma seção onde mais de uma variável de um único tipo podem ser definidas ao mesmo tempo, separando-as por uma vírgula, e variáveis de diferentes tipos podem ser declaradas dentro da mesma seção de declaração, separando-se as declarações com um ponto e vírgula. As variáveis podem ser redeclaradas ao longo da especificação, passando a assumir o outro tipo a partir do ponto onde foram redeclaradas. Abaixo, exemplifica-se a declaração de algumas variáveis globais de dois tipos diferentes em uma mesma declaração. A variável `c` possui o tipo `S * S` após a execução da declaração.

```
var a,c: S;
    b,c: S * S
```

Variáveis locais são definidas pela palavra-chave `forall` e introduzidas antes dos axiomas, separando-se dos mesmos por um ponto final. Por exemplo, o axioma

```
.forall z,r: T . r = z
```

define as variáveis `z` e `r`, ambas do tipo `T` e as utiliza em uma igualdade. Após o axioma, ambas as variáveis não estão mais presentes no contexto.

Uma função sem parâmetros, também chamada constante, pode ser definida pelas palavras-chave `op` ou `fun`. Embora as duas palavras-chave definam funções, elas apresentam comportamento diferente com respeito aos subtipos, como será visto mais adiante. Uma constante `c` do tipo `S` pode ser definida por uma das seguintes declarações:

```
op c: S
fun c: S
```

Ao substituir o tipo da constante por um tipo de função (total ou parcial), cria-se uma declaração de função com um parâmetro, como na declaração a seguir, que define uma função `f` que deve ser aplicada a um termo do tipo `S`, resultando em um termo do tipo `T`:

```
fun f: S -> T
```

A substituição indutiva de tipos por tipos de função dá origem a funções de mais de um parâmetro. Se o tipo `S` na função `f` anterior for substituído pelo tipo de função `S -> S`, define-se uma nova função que recebe dois parâmetros do tipo `S` e retorna um termo do tipo `T`, como visto na declaração da função `g` a seguir:

```
fun g: S -> S -> T
```

Uma função é aplicada aos seus parâmetros por justaposição, ou seja, a aplicação da função `g: S -> S -> T` aos parâmetros `a: S` e `b: S` resulta no termo `g a b: T`, que representa um termo do tipo `T`. A associação de termos na definição de tipos ocorre à direita e a associação de termos na aplicação de funções ocorre à esquerda. Assim, o tipo da função `g: S -> S -> T` é interpretado internamente como `g: (S -> (S -> T))` e a aplicação da função, `g a b: T`, é interpretada como `((g a) b): T`.

Funções definidas com tipos de função permitem a aplicação parcial de funções. A aplicação da função `g: S -> S -> T` a um parâmetro `a: S` resulta no termo `g a:S -> T`, que representa uma função que recebe um parâmetro do tipo `S` e retorna um termo do tipo `T`. A aplicação da função `g a: S -> T` a um parâmetro `b: S` origina o termo `g a b: T`.

As funções podem ser sobrecarregadas bastando-se definir um novo perfil para um mesmo nome de função. A função `g` anteriormente definida pode ser sobrecarregada com um perfil que utilize produtos de tipos da seguinte forma:

```
fun g: S * S -> T
```

Uma outra forma de definir o tipo de funções consiste em usar produto de tipos. Nesta forma de definição, é possível definir a posição dos parâmetros em relação ao nome da função. Para tanto, indica-se a posição de cada parâmetro através do posicionador `__`, formado por dois caracteres sublinhado seguidos.

Os termos de um tipo definido com produto de tipos devem ser construídos na forma de tuplas. Desta forma, um elemento do tipo $(t_1 * \ldots * t_n)$ deve ser escrito sob a forma $(s_1 : t_1, \ldots, s_n : t_n)$. A tupla vazia, `()` é o elemento (único) do tipo unitário `Unit`.

A seguir, ilustram-se alguns perfis de funções e as respectivas formas de aplicação da função sobre variáveis, sem e com as respectivas anotações de tipos.

- Declaração de variáveis:

  ```
  type S,T
  var a,b: S
      x: T
  ```

- Função definida com tipos de função:

  ```
  fun g1: S -> S -> T
  . g1 a b = x
  . (g1: S -> S -> T) (a:S) (b:S) = (x:T)
  ```

- Função definida com produto de tipos sem posicionador de variáveis:

  ```
  fun g2: S * S -> T
  . g2 (a, b) = x
  . (g2: S * S -> T) (a:S, b:S) = (x:T)
  ```

- Função prefixa definida com produto de tipos:

```
fun g3 __ __: S * S -> T
. g3 a b = x
. (g3 __ __: S * S -> T) (a:S, b:S) = (x:T)
```

- Função infixa definida com produto de tipos:

```
fun __gi__: S * S -> T
. a gi b = x
. (__gi__: S * S -> T) (a:S, b:S) = (x:T)
```

- Função pós-fixada definida com produto de tipos:

```
fun __ __ gp: S * S -> T
. a b gp = x
. (__ __ gp: S * S -> T) (a:S, b:S) = (x:T)
```

As funções parciais são avaliadas de forma estrita, ou seja, todos os parâmetros são avaliados antes que a aplicação da função seja avaliada. Isto pode ser descrito pelo axioma `def f(a) => def a`, ou seja, a definição da aplicação de uma função sempre implica que seus parâmetros estão definidos.

Tipos com avaliação preguiçosa podem ser simulados em ambientes de avaliação estrita. Para simular a avaliação preguiçosa de um tipo `s`, basta movê-lo para o tipo `Unit ->? s`. Assim, obtém-se funções com tipos com avaliação preguiçosa tais como `?s -> ?t`.

Há duas regras para a aplicação de funções com avaliação preguiçosa e para a aplicação de funções com avaliação estrita sobre termos de tipos com avaliação preguiçosa, a saber:

- Sejam os termos `a: ?S -> ?T` ou `a: ?S -> ?T` e o termo `b: S`. A aplicação de `a` sobre `b` resulta no termo `a b: t`, no qual o termo `b` é implicitamente substituído por `\ . b`;

- Sejam os termos `a: S -> ?T` ou `a: S -> T` e o termo `b: ?S`. A aplicação de `a` sobre `b` resulta no termo `a b: t`, no qual o termo `b` é implicitamente substituído por `b ()`;

Funções parciais sobre o tipo `Unit` podem ser interpretadas como predicados (ou fórmulas), onde a definição do predicado corresponde à satisfação da função. Para facilitar a declaração de predicados, o sinônimo de tipos `Pred s := s ->? Unit` foi criado. O operador de igualdade interna `=e=: Pred (T * T)`, onde `T` é um tipo, é um exemplo de função parcial interpretada como predicado.

As assertivas de definição são fórmulas atômicas na forma `def t`, onde t é um termo, e servem para verificar se o termo em questão está ou não definido. Por exemplo, a fórmula `def t => t = x` indica que a igualdade entre os termos `t` e `x` ocorre sempre que o termo `t` está definido.

Há duas formas de igualdade entre termos. Uma equação universal `a = b` é verdadeira quanto ambos os termos estão definidos e são iguais e, também, quando ambos os termos são indefinidos. Uma equação existencial `a =e= b`, é verdadeira apenas quando ambos os termos estão definidos e são iguais. A relação entre as duas igualdades pode ser resumida pelo axioma `a =e= b <=> def a /\ def b /\ a = b`, onde o símbolo `<=>` denota equivalência entre termos e o símbolo `/\` representa a conjunção entre termos. Pode-se verificar, então, que a assertiva de definição `def a` equivale ao termo `a =e= a`.

A formação da λ-abstração parcial que leva uma variável `s: S` a um termo `t: T` pode ser escrita na forma `\ s:S . t:T`. Caso o termo `t` esteja definido para todos os possíveis valores da variável `s`, pode-se construir uma λ-abstração total na forma `\ s:S .! t:T`. Se o termo `t` não estiver definido para todos os valores de `s`, a λ-abstração total, embora sintaticamente correta, não representa um valor válido. Em oposição, a λ-abstração parcial sempre está definida.

A aplicação de sucessivas λ-abstrações podem ser combinadas na forma `\ x y z . t` simplificando o termo `\ x .! \ y .! \ z . t`, onde `t` é um termo. A abstração sobre uma variável não utilizada do tipo `Unit` pode ser escrita na forma `\ . t`.

A associação local de variáveis pode ser escrita na forma `let x = t in z`, onde `t` e `z` são termos e `x` é uma variável. Uma outra forma equivalente pode ser usada, a saber: `z where x = t`. Associações consecutivas podem ser realizadas listando-se as mesmas com um separador, na forma: `let x1 = t1; x2 = t2; ... ; xn = tn in z`.

É possível utilizar casamento de padrões para associação de variáveis como nas linguagens funcionais de programação. Variáveis podem ser associadas por casamento de padrões a componentes de tuplas ou a construtores de tipos e podem ser arbitrariamente encadeados. Por exemplo, pode-se utilizar casamento de padrões em associações locais de variáveis, como em `let (x,y) = (t,z) in x`, onde a variável `x` é associada ao termo `t` e a variável `y` é associada ao termo `z`. Como a linguagem não possui funções de projeção para elementos de tuplas, a maneira padrão de ter acesso a esses elementos é através do casamento de padrões.

A associação anteriormente descrita, `let (x,y) = (t,z) in x`, também pode ser escrita pelo atalho sintático `t res z`. O termo `t res z` está definido se, e somente se, `t` e `z` estão definidos e, neste caso, são iguais a `t`. Um caso particular desta expressão ocorre quando o termo `z` for um predicado; neste caso, a expressão estará definida se, e somente se, `t` estiver definido e o predicado `z` valer.

HasCasl permite polimorfismo através de classes de tipos, permitindo que tipos e funções dependam de variáveis de tipos (incluindo variáveis de construtores de tipos) e que axiomas sejam universalmente quantificados sobre tipos no nível mais externo da abstração. As variáveis de tipo podem assumir qualquer tipo declarado na especificação, podendo-se restringir o intervalo de valores a uma classe de tipos, como em Haskell.

O Universo de tipos (*kinds*) em HasCasl é formado pela gramática

```
K ::= C | K -> K
```

onde `K` é o conjunto de tipos (*kinds*) e `C` é o conjunto de classes, no qual está contida a classe `Type` representando o universo de todos os tipos.

Os tipos (*kinds*) da forma `K1 -> K2` são chamados tipos (*kinds*) construtores e um tipo (*kind*) é chamado primitivo (*raw*) se utilizar apenas a classe `Type` em sua formação. A relação de subclasse é uma relação entre classes e tipos (*kinds*) tal que cada classe de equivalência de uma dada classe `Cl` gerada pela relação de subclasse possui apenas um tipo (*kind*) primitivo, indicado por `raw(Cl)` e denotado tipo (*kind*) primitivo da classe `Cl`. Isto significa que cada tipo (*kind*) `K` é equivalente a um único tipo (*kind*) primitivo, denotado `raw(K)`, o qual pode ser obtido substituindo-se todo tipo (*kind*) de `K` pelo respectivo tipo (*kind*) primitivo.

Dessa forma, a classe `Type` é o universo de todos os tipos, os tipos (*kinds*) construtores são o universo dos construtores de tipos e as classes são subconjuntos desse universos de acordo com as regras da relação de subclasse.

Declara-se uma classe `C` como subclasse de um tipo (*kind*) `K` pela sentença `class C < K`. Uma classe pode ser declarada subclasse de várias outras classes desde que todas possuam o mesmo tipo (*kind*) primitivo. Subclasses dos tipos (*kinds*) construtores são chamadas classes construtoras e classes declaradas sem um supertipo (*superkind*) explicito são consideradas subclasses da classe `Type`. Uma classe pode ser declarada subclasse de várias classes, desde que todas possuam o mesmo tipo primitivo.

Variáveis de tipo podem ser declaradas com o seu respectivo tipo (*kind*) da mesma forma que variáveis comuns são declaradas. Estas variáveis são utilizadas no lugar de tipos ou construtores de tipos tornando as entidades (tipos, funções ou axiomas) onde são utilizadas polimórficas sobre o tipo (*kind*) da variável. Com o uso de variáveis de tipo, pode-se obter construtores de tipos, da forma:

```
type t p1 .. pn: K
```

onde `p1 ... pn` são variáveis de tipo com tipos (*kinds*) `K1, ... , Kn`, respectivamente. Tal declaração introduz um construtor de tipo `t` do tipo `K1 -> ... -> Kn -> K`. Em particular, quando o construtor não possui variáveis, seu tipo é `K`.

Funções polimórficas são rotuladas com esquemas de tipo onde os tipos são quantificados sobre as variáveis de tipo no nível mais externo da declaração. Os axiomas polimórficos, por sua vez, são universalmente quantificados de forma implícita sobre as variáveis não associadas (varáveis livres).

Os construtores de tipo padrão compreendem os construtores para tipos de função e de produto de tipos – `*`, `->` e `->?` que possuem tipo (*kind*) `Type -> Type -> Type` – e construtor do tipo unitário `()`, que possui tipo (*kind*) `Type`. Os demais construtores de tipos são todos definidos pelo usuário.

Construtores de tipos são entidades diferentes dos sinônimos de tipos parametrizados. A declaração

```
var a: Type
type DList a := List(List a)
```

introduz o sinônimo de tipo parametrizado `DList`, com tipo (*kind*) `Type -> Type`, que não deve ser confundido com um possível construtor de tipo.

Os tipos (*kinds*) são estruturas sintáticas apenas, não possuindo nenhuma semântica associada. O conjunto de tipos e subtipos derivado de um tipo (*kind*) é governado pelos tipos (*kinds*) associados aos construtores de tipos e pelas relações de subclasse. Dessa forma,

```
var a: Ord
type List a, Nat: Ord
```

declara o tipo `Nat` como pertencente à classe `Ord` e o construtor de tipo `List a` como tendo o tipo (*kind*) `Ord -> Ord`, ou seja, o tipo `List t` pertence à classe `Ord` sempre que o tipo `t` pertencer à classe `Ord`.

Embora funções e axiomas não façam parte da definição de uma classe, os mesmos podem ser associados a uma classe através de um bloco identificado por parênteses após a declaração da classe. Esta declaração funciona como uma interface para a classe. A seguir, transcreve-se um trecho da especificação `Ord` (ver especificação completa no Apêndice A.4, na página 91), que mostra a declaração da função `__<__` com seus axiomas dentro da interface da classe `Ord`:

```
class Ord < Eq
{
 var a: Ord
 fun __<__ : a * a -> Bool
 var    x, y, z, w: a
 . (x == y) = True => (x < y) = False                        %(LeIrreflexivity)%
 . (x < y) = True => y < x = False                           %(LeTAsymmetry)% %implied
 . (x < y) = True /\ (y < z) = True => (x < z) = True      %(LeTTransitive)%
 . (x < y) = True \/ (y < x) = True \/ (x == y) = True     %(LeTTotal)%
```

Subclasses ou tipos de dados que devam obedecer à interface podem ser declarados como instâncias de uma classe com o uso da palavra-chave `instance`. No primeiro caso, ela é incluída na declaração de uma classe, entre a palavra-chave `class` e a definição de subclasse, na forma `class instance subclass_name < class_name`. No segundo caso, ela é incluída entre a palavra-chave `type` e a declaração do tipo, na forma `type instance type_name : class_name`. A declaração inclui, ainda, axiomas que definam o comportamento de funções da classe sobre variáveis de tipo da subclasse ou do tipo que estão sendo declarados. Tal declaração gera uma obrigação de prova que garante que os axiomas da interface decorrem dos axiomas da subclasse ou do tipo que foram declarados instâncias da classe. O tipo de dado `Ordering` é declarado instância da classe `Ord` e são adicionados axiomas relativos ao comportamento da função definida anteriormente em relação ao tipo de dados, como visto à seguir:

```
type instance Ordering: Ord
. (LT < EQ) = True                    %(IO013)%
. (EQ < GT) = True                    %(IO014)%
. (LT < GT) = True                    %(IO015)%
```

HASCASL suporta polimorfismo sobre tipos de ordens superiores. A especificação `Monad`, da biblioteca da linguagem CASL, serve de exemplo para a especificação de classes construtoras e construtores de tipos de ordens superiores. No momento, a tradução de especificações com este tipo de polimorfismo para a linguagem do provador de teoremas ainda não é suportada.

3.5 Especificações Estruturais

Sistemas complexos possuem vários componentes com lógicas complexas. A modularização destes sistemas é uma peça fundamental para facilitar a sua manutenção. As especificações básicas definem a lógica dos sistemas, criando módulos auto-contidos que expressam um componente ou uma funcionalidade. Por sua vez, as especificações estruturais permitem unir ou estender estes módulos, renomeando ou ocultando símbolos enquanto especificações complexas são construídas.

Pode-se unir especificações básicas através do conectivo `and`. A união de especificações é associativa e comutativa, ou seja, a especificação resultante independe da ordem em que as especificações são unidas. Por exemplo, as especificações `Spec1`, `Spec2` e `Spec3` podem ser unidas para formar a especificação `Spec4` da seguinte forma:

```
spec Spec1 = ...
```

```
end
spec Spec2 = ....
end
spec Spec3 = ...
end
spec Spec4 =
        Spec1 and Spec2 and Spec3
end
```

Especificações básicas também podem ser estendidas com a inclusão de novos símbolos. Para tanto, utiliza-se o conectivo `then`, que pode ser aplicado mais de uma vez na definição de uma mesma especificação, separando a mesma em subespecificações. As subespecificações diferem de especificações por serem criadas localmente, ou seja, elas não são nomeadas e, dessa forma, não podem ser referenciadas. Apenas a especificação formada por todas as extensões recebe um nome e pode ser estendida ou unida à outras especificações. A seguir, estende-se uma especificação `Spec1` com o conteúdo da especificação `Spec2`. A subespecificação resultante é estendida com a especificação `Spec3`, formando, finalmente, a especificação `Spec4`.

```
spec Spec4 =
        Spec1
then
        Spec2
then
        Spec3
end
```

É comum utilizar as duas operações conjuntamente ao se definir novas especificações que utilizem símbolos de outras especificações previamente definidas. Isto pode ser verificado em quase todas as especificações da biblioteca apresentada no Apêndice A, na página 88, uma vez que existe dependência entre as várias especificações que passam a utilizar tipos e funções definidos em especificações anteriores.

Por padrão, ambas as operações exportam todas as funções e tipos definidos para as especificações resultantes (variáveis não são consideradas símbolos e não são exportadas). Dessa forma, todos os tipos com mesmo nome serão tratados como sendo o mesmo tipo e funções com mesmo nome mas perfis diferentes serão automaticamente sobrecarregadas. Ambas as operações podem combinar especificações de qualquer um dos modelos semânticos existentes (*loose*, *generated* ou *free*), misturando-os sem perda de propriedades.

Em alguns casos, especificações combinadas podem exportar símbolos com o mesmo nome e perfil, mas que possuem comportamentos diferentes, representando entidades diferentes. É possível renomear os símbolos (tipos e funções) existentes em uma especificação através da palavra-chave `with`, com o auxílio do símbolo `|->` para indicar a renomeação de cada símbolo. O exemplo a seguir mostra a criação da especificação `Bool` a partir da especificação `Boolean`, que foi importada da biblioteca de CASL. A palavra-chave `with` introduz a seção de renomeação de símbolos e o uso das palavras-chave `sort` e `op` é opcional, embora seu uso ajude na leitura da especificação.

```
from Basic/SimpleDatatypes get Boolean
spec Bool = {Boolean with
         sort Boolean |-> Bool,
         op Not__ |-> not__,
         op __And__ |-> __&&__,
         op __Or__ |-> __||__
        }
then
op otherwise: Bool
. otherwise = True
```

A renomeação de um tipo também é realizada nos perfis das funções importadas, sejam elas renomeadas ou não, de forma automática. Pode-se apenas indicar a origem de um dado símbolo, sem efetivamente renomeá-lo, bastando apenas citar o seu nome após a palavra-chave `with` ou renomeando-o para o mesmo nome atual. Esta indicação permite deixar claro a existência de símbolos sobrecarregados ao indicar que um mesmo símbolo provêm de duas ou mais especificações que estão sendo unidas ou estendidas.

Ainda é possível ocultar símbolos que se deseje manter local a uma determinada especificação. Este é o caso de funções auxiliares criadas para facilitar as especificações de funções mais complexas e que não devem ser utilizadas por outras especificações que venham a estender ou utilizar a especificação atual em uma união. Pode-se ocultar símbolos de uma especificação escolhendo-se os símbolos a serem escondidos ou escolhendo-se os símbolos que se deseje exportar. Para tanto, são utilizadas as palavras-chave `hide` e `reveal`, respectivamente, após a definição de uma especificação. Na especificação a seguir, duas funções auxiliares (`sum'` e `product'`) foram ocultadas da especificação.

```
spec ListWithNumbers = ListNoNumbers and NumericClasses then {
     vars a,b: Type;
          c,d: Num;
          x,y : a;
          xs,ys : List a;
          n,nx : Int;
          z,w: Int;
          zs,ws: List Int
     fun length: List a -> Int;
     fun take: Int -> List a -> List a
     fun drop: Int -> List a -> List a
     fun splitAt: Int -> List a -> (List a * List a)
     fun sum: List c -> c
     fun sum': List c -> c -> c
     fun product: List c -> c
```

```
     fun product': List c -> c -> c
...
} hide sum', product'
end
```

Há ainda uma terceira facilidade para transformar símbolos em símbolos locais. Esta facilidade equivale a definir uma especificação e depois ocultar símbolos explicitamente com o uso da palavra-chave `hide`. Sua vantagem é tornar implícito o processo de ocultar os símbolos, indicando que os mesmos devem ser locais. A construção consiste em definir os símbolos locais entre as palavras-chave `local` e `within`, seguidos dos símbolos que serão exportados, da seguinte forma:

```
spec Spec1 =
local
        sort tipoLocal
        op operacaoLocal: ...
within
        sort tipoExportado
        op operacaoExportada
end
```

3.6 Bibliotecas de Especificações

A criação de bibliotecas é um dos requisitos para o reuso de código. O suporte a bibliotecas em CASL inclui suporte a bibliotecas locais e distribuídas, com versionamento. A linguagem permite, ainda, anotações de precedência e associatividade de operadores e anotações que indicam a apresentação de operadores nos formatos LaTeX, *HTML* e *RTF*.

Uma biblioteca local é uma coleção auto-contida e nomeada de especificações. A visibilidade de especificações é linear, exigindo a declaração de especificações antes que possam ser referenciadas. Todos os demais tipos de especificação podem ser agregados em bibliotecas de forma a permitir que sejam importados por novas especificações.

As bibliotecas podem ser colocadas à disposição em servidores, através dos quais podem ser remotamente acessadas, quando necessário. Para tanto, basta que elas possuam uma *URL* que obedeça a uma hierarquia de pastas através da qual se consiga alcançar a biblioteca desejada de forma única.

Além dos diferentes tipos de especificação, as bibliotecas possuem sintaxe específica para indicar o nome pelo qual serão referenciadas, o nome dos autores e a data de criação ou modificação. Estas duas últimas anotações permitem a declaração de mais de um item e podem se referir a toda a biblioteca, quando colocadas no começo do arquivo, ou a uma particular especificação, quando inseridas antes da especificação em questão. A sintaxe para essas anotações é ilustrada pelo exemplo fictício abaixo.

```
library Diretório/Subdiretório/NomeDaBiblioteca
%authors( Autor1 < autor1@host> , Autor2 <autor2@host> )%
%dates 25 Jan 2009, 25 Nov 2009

spec Spec1 = ...

%autors Autor3 <autor3@host>
%dates 25 Ago 2009
spec Spec2 = ...
```

No exemplo, é criada uma biblioteca com o nome `NomeDaBiblioteca`, e que reside na estrutura de diretório `${HETS_LIB}/Diretório/Subdiretório`, onde `{HETS_LIB}` é uma variável de ambiente do sistema operacional que indica o caminho do repositório local das bibliotecas da ferramenta HETS, a qual é responsável por analisar as especificações. Caso a biblioteca seja distribuída, a estrutura de diretórios deverá ser substituída pela *URL* com o caminho remoto completo de onde a biblioteca `NomeDaBiblioteca` será encontrada.

Dois autores e duas datas, sendo uma delas de revisão, são associados à biblioteca. Logo abaixo, um terceiro autor e uma nova data são associados à uma particular especificação.

Operações podem ser associadas a caracteres específicos em um determinado ambiente de exibição. Atualmente, pode-se indicar como um operador deve ser exibido em arquivos de tipos LaTeX, *HTML* e *RTF*. A indicação para cada tipo de ambiente é opcional, podendo-se indicar a exibição em todos ou em apenas alguns ambientes. Quando não houver indicação para um ambiente, o nome da operação será utilizado naquele ambiente, como seria esperado. O exemplo a seguir define que a operação `__<=__` será exibida com o uso do caractere '$\leq$' em arquivos LaTeX. As anotações para *HTML* e *RTF* seriam feitas adicionando-se as anotações `%HTML` e `%RTF`, respecticamente, após a anotação `%LATEX`.

```
%display __<=__     %LATEX __\leq__
```

As anotações que permitem indicar a precedência de operadores e a sua associatividade são mostradas a seguir. No exemplo, as funções `__op1__` e `__op2__` são definidas com uma precedência menor em relação à função `__op3__`, ou seja, o termo `a op1 (b op3 c)`, onde `a`, `b` e `c` são variáveis, pode ser escrito na forma `a op1 b op3 c`. Já as funções `__op1__` e `__op3__` são declaradas associativas à esquerda, ou seja, o termo `a op1 b op1 c` é equivalente ao termo `(a op1 b) op1 c`.

```
%prec {__op1__, __op2__} < {__op3__}
%left_assoc __op1__ , __op3__
```

Números na linguagem CASL são definidos com o auxílio de anotações. Os dígitos são definidos como constantes e são concatenados através da operação `__@@__`, definida na especificação `Nat` da biblioteca de CASL. O conjunto de anotações a seguir permite que os números sejam escritos na forma comum que sejam tranformados na concatenação de caracteres de forma transparente. Dessa forma, pode-se utilizar o inteiro "145" e as ferramentas da linguagem farão a tradução do mesmo para o termo "1@@4@@5".

```
%left_assoc __@@__
%number __@@__
```

Ainda para ajudar na definição de números, as anotações a seguir permitem o uso de números de ponto flutuante com o separador '.', como em 2743., e de números com expoentes, como em `10E-12`. Os números serão convertidos automaticamente para `2:::7@@4@@3` e `1@@0E-1@@2`, respectivamente. As funções `__:::__` e `__E__` também estão definidas nas especificações numéricas da biblioteca da linguagem Casl.

```
%floating __:::__, __E__
%prec {__E__} < {__:::__}
```

Para que as especificações de uma biblioteca distribuída possam ser utilizadas, é necessário que as mesmas sejam importadas pelo arquivo onde serão utilizadas. Também é possível renomear uma especificação no momento de sua importação, com uma sintaxe parecida à da renomeação de símbolos em especificações. A sintaxe para a importação de especificações é ilustrada a seguir, onde são importadas as especificações `Spec1` e `Spec2` da biblioteca fictícia criada anteriormente, com a renomeação da especificação `Spec2` para `Spec4`.

```
from Diretório/Subdiretório/NomeDaBiblioteca get Spec1, Spec2 |-> Spec4
```

O uso de versões permite que mudanças sejam introduzidas em especificações sem que as primeiras alterem o comportamento de códigos que importavam uma versão anterior das bibliotecas. Para tanto, é possível indicar a versão da biblioteca tanto na sua declaração quando no momento de importá-la, indicando qual versão da biblioteca se deseja importar e permitindo que várias versões coexistam no mesmo repositório. A indicação de versão é realizada pela palavra-chave `version` seguida do número associado à versão. Esta indicação deve ser incluída na declaração e na importação de uma biblioteca seguindo a sintaxe:

```
library Diretório/Subdiretório/NomeDaBiblioteca version 1.0
...
from Diretório/Subdiretório/NomeDaBiblioteca version 1.0 get Spec1
```

Embora seja possível utilizar as anotações para importar especificações em qualquer local de uma biblioteca, deve-se preferir incluir todas as anotações no começo do arquivo de forma a tornar clara a visualização das dependências da biblioteca sendo especificada.

Capítulo 4

Especificação de uma Biblioteca para a Linguagem HasCASL

4.1 Considerações iniciais

Para capturar todas as características da linguagem Haskell, a biblioteca deveria utilizar os conceitos de avaliação preguiçosa e funções contínuas, permitindo tipos de dados infinitos. O uso de tais conceitos exigiria o emprego das construções mais avançadas da linguagem HasCasl e um profundo conhecimento prévio da linguagem HOL, para que pudessem ser escritas as provas dos teoremas gerados a serem verificados pelo provador Isabelle. Tais exigências inviabilizaram o uso destes conceitos no primeiro contato com a metodologia de especificação algébrica utilizada no projeto. Decidiu-se, então, iniciar a especificação com tipos de dados com avaliação estrita e, em um refinamento posterior, incluir o uso de tipos com avaliação preguiçosa.

A decisão anterior exigiu que os tipos de dados fossem totalmente reescritos. Com isso, uma especificação foi criada para representar o tipo booleano. Desta forma, não foi mais possível utilizar o tipo de dados padrão para *verdadeiro* e *falso*, e nem seus operadores para negação, conjunção e disjunção. Isto resultou na necessidade de se comparar o resultado de cada sentença com os construtores do novo tipo booleano, tornando a sintaxe visualmente carregada. Para eliminar estas comparações, seria necessário que o tipo booleano fosse definido a partir do tipo `Unit`, de Casl, com avaliação preguiçosa. Dessa forma, tal eliminação foi, também, deixada para um refinamento posterior.

A nomenclatura de tipos e funções da biblioteca Prelude foi utilizada sempre que possível. Ao importar um tipo de dado já definido na biblioteca da linguagem Casl, o mesmo teve seu nome alterado para o respectivo nome na biblioteca Prelude através do uso da sintaxe apropriada presente em HasCasl. Quando novas funções foram especificadas, utilizou-se o nome empregado na biblioteca Prelude sempre que possível. No entanto, em alguns casos o nome utilizado na biblioteca Prelude conflitava com palavras reservadas da linguagem HasCasl; neste caso, o sufixo H foi adicionado ao nome.

As especificações escritas em HasCasl são orientadas a propriedades, ou seja, as especificações modelam um problema e propriedades são verificadas através da criação e verificação de teoremas. As especificações são definidas por operadores e predicados, que juntos constituem uma assinatura, e axiomas que governam os elementos da assinatura. As propriedades da especificação que se deseja verificar são incluídas como teoremas a serem provados utilizando-se um provador de teorema.

As provas escritas em HOL para Isabelle frequentemente requerem que axiomas sejam reescritos de forma que possam ser utilizados pelo provador em operações automáticas. Tais axiomas devem ser reescritos na forma de lemas e devem ser provados da mesma forma que os teoremas. Alguns dos lemas foram adicionados diretamente às especificações, sendo transformados em teoremas pela ferramenta Hets; outros, no entanto, foram escritos diretamente em HOL, por questões de praticidade. Todos os teoremas, lemas e axiomas foram nomeados para facilitar o uso dos mesmos nas provas geradas para o Isabelle.

Operadores, tais como $+$, $-$, $*$, $/$, $<$, entre outros, são definidos em CASL e Haskell de forma infixa e com tipos *curried*. Em Haskell, no entanto, é possível transformar um operador em uma função com tipos *uncurried*, bastando colocar o operador entre parênteses. A função resultante recebe o nome de seção, em Haskell. Para obter o mesmo efeito em HASCASL, é preciso criar uma λ-abstração envolvendo o operador, de forma que esta possa ser passada como parâmetro. Este mecanismo foi utilizado para passar os operadores $<$ e $==$ como parâmetros para funções de segunda ordem que operam sobre listas.

O capítulo apresenta, a seguir, a descrição de cada uma das especificações em uma seção específica. O código completo das especificações foi agrupado no Apêndice A, na página 88 para que possa ser facilmente consultado.

4.2 Bool, a primeira especificação

Inicialmente, o tipo de dado `Bool`, representando o tipo booleano com construtores para *verdadeiro* e *falso*, foi criado através da importação do tipo de dado `Boolean` da biblioteca de CASL, renomeando-se o tipo e as funções para os nomes utilizados na biblioteca Prelude, como visto a seguir:

```
from Basic/SimpleDatatypes get Boolean
spec Bool = {Boolean with
         sort Boolean |-> Bool,
         op Not__ |-> not__,
         op __And__ |-> __&&__,
         op __Or__ |-> __||__
        }
then
op otherwise: Bool
```

```
. otherwise = True
```

Tendo em vista o refinamento para inclusão de tipos com avaliação preguiçosa, decidiu-se recriar o tipo de dado deste o início, uma vez que o tipo importado da biblioteca Casl possuía apenas tipos com avaliação estrita. Foram adicionados teoremas que, embora simples, eram necessários para simplificar provas geradas para o Isabelle. A especificação final pode ser vista abaixo:

```
spec Bool = %mono
free type Bool ::= True | False
fun Not__ : Bool -> Bool
fun __&&__ : Bool * Bool -> Bool
fun __||__ : Bool * Bool -> Bool
fun otherwiseH: Bool
...
end
```

4.3 *Eq*: especificando a relação de equivalência

Classes em HasCasl são similares às de Haskell. A declaração de uma classe inclui funções e axiomas sobre variáveis de tipo da classe em questão, representando a interface desta classe. Uma declaração de instância de tipo obriga um tipo a pertencer a uma classe. Isto significa que o tipo deve obedecer a todas as declarações de funções da interface da classe, assim como a todos os axiomas.

A especificação `Eq` (ver Apêndice A.3, na página 90), para funções de igualdade e desigualdade, iniciou o uso de classes. A declaração da classe `Eq`, a qual inclui uma função de igualdade e axiomas para simetria, reflexividade e transitividade desta operação, pode ser vista a seguir.

```
spec Eq = Bool then
class Eq {
var a: Eq
fun __==__ : a * a -> Bool
fun __/=__ : a * a -> Bool
vars x,y,z: a
. x = y => (x == y) = True                    %(EqualTDef)%
. x == y = y == x                             %(EqualSymDef)%
. (x == x) = True                             %(EqualReflex)%
. (x == y) = True /\ (y == z)
    = True => (x == z) = True                 %(EqualTransT)%
. (x /= y) = Not (x == y)                     %(DiffDef)%
...
}
type instance Bool: Eq
...
. (False == True) = False                     %(IBE3)%
...
type instance Unit: Eq
...
end
```

A função de desigualdade é definida usando-se a função de igualdade. Evitou-se definir a equivalência entre a função de igualdade definida e a função de igualdade padrão de HasCasl para permitir com que cada tipo de dado definisse o escopo da igualdade. Foram criados axiomas para verificar o comportamento da função de diferença e teoremas para auxiliar no processo de provas geradas para o Isabelle.

Os tipos `Bool` e `Unit` foram declarados como instâncias da classe `Eq`. Foi necessário axiomatizar a negação da igualdade entre dois construtores diferentes do tipo `Bool` porque a igualdade da classe não é equivalente à igualdade padrão de HasCasl, e é esta última a igualdade utilizada para criar axiomas entre os construtores dos tipos declarados como `free`. Os demais teoremas puderam ser provados a partir do axioma anterior. Não foi necessário incluir nenhum axioma na declaração envolvendo o tipo `Unit` porque este tipo possui apenas um construtor, e as provas decorrem diretamente dos axiomas da classe.

4.4 A Especificação *Ord*, para Ordenação

A próxima especificação criada foi a de relações de ordenação, nomeada `Ord`. Inicialmente, tentou-se importar a especificação `Ord` presente na biblioteca de CASL em HASCASL/Metatheory/Ord. No entanto, a especificação existente faz uso de avaliação preguiçosa e, desta forma, não poderia ser importada. A especificação, então, foi escrita desde o início (ver Apêndice A.4, na página 91).

Para criar a especificação `Ord`, o tipo de dado `Ordering` foi especificado e declarado como instância da classe `Eq`. Novamente, foi preciso axiomatizar a negação da igualdade entre os três construtores do tipo, dois a dois.

```
spec Ord = Eq and Bool then
free type Ordering ::= LT | EQ | GT
type instance Ordering: Eq
. (LT == EQ) = False  %(IOE04)%
. (LT == GT) = False  %(IOE05)%
. (EQ == GT) = False  %(IOE06)%
```

A classe `Ord` foi criada como subclasse da classe `Eq`, como em Haskell. Especificou-se uma relação de ordenação total (`__<__`), com axiomas para irreflexividade e transitividade, um teorema para assimetria, decorrente dos axiomas anteriores, e um axioma para totalidade, como visto a seguir.

```
class Ord < Eq
{
```

```
 var a: Ord
 fun compare: a -> a -> Ordering
 fun __<__ : a * a -> Bool
 fun __>__ : a * a -> Bool
 fun __<=__ : a * a -> Bool
 fun __>=__ : a * a -> Bool
 fun min: a -> a -> a
 fun max: a -> a -> a
 var    x, y, z, w: a
. (x == y) = True => (x < y) = False                    %(LeIrreflexivity)%
. (x < y) = True => y < x = False                 %(LeTAsymmetry)% %implied
. (x < y) = True /\ (y < z) = True
=> (x < z) = True                                        %(LeTTransitive)%
. (x < y) = True \/ (y < x) = True
\/ (x == y) = True                                           %(LeTTotal)%
```

As demais funções de ordenação foram definidas usando-se a relação de ordenação total e a relação de equivalência, quando aplicável, como visto abaixo. As propriedades de irreflexividade, assimetria, transitividade e totalidade foram definidas como teoremas.

```
 . (x > y) = (y < x)                                             %(GeDef)%
 . (x <= y) = (x < y) || (x == y)                               %(LeqDef)%
 . (x >= y) = ((x > y) || (x == y))                             %(GeqDef)%
```

Os axiomas a seguir foram definidos para relacionar as quatro funções de ordenação e a função de equivalência entre sim, para ambos os construtores do tipo de dado `Bool`. Adicionalmente, foram criados teoremas para verificar que as funções de ordenação se comportam da maneira esperada.

```
%% Relates == and ordering
 . (x == y) = True <=> (x < y) = False /\ (x > y) = False       %(EqTSOrdRel)%
```

```
. (x == y) = False <=> (x < y) = True \/ (x > y) = True        %(EqFSOrdRel)%
. (x == y) = True <=> (x <= y) = True /\ (x >= y) = True        %(EqTOrdRel)%
. (x == y) = False <=> (x <= y) = True \/ (x >= y) = True       %(EqFOrdRel)%
. (x == y) = True /\ (y < z) = True => (x < z) = True       %(EqTOrdTSubstE)%
. (x == y) = True /\ (y < z) = False => (x < z) = False     %(EqTOrdFSubstE)%
. (x == y) = True /\ (z < y) = True => (z < x) = True       %(EqTOrdTSubstD)%
. (x == y) = True /\ (z < y) = False => (z < x) = False     %(EqTOrdFSubstD)%
. (x < y) = True <=> (x > y) = False /\ (x == y) = False     %(LeTGeFEqFRel)%
. (x < y) = False <=> (x > y) = True \/ (x == y) = True      %(LeFGeTEqTRel)%
```

Finalizando a definição da classe, foram definidas as funções de comparação, máximo e mínimo, como visto a seguir.

```
%% Definitions to compare, max and min using relational operations.
 . (compare x y == LT) = (x < y)                                  %(CmpLTDef)%
 . (compare x y == EQ) = (x == y)                                 %(CmpEQDef)%
 . (compare x y == GT) = (x > y)                                  %(CmpGTDef)%
%% Define min, max
 . (max x y == y) = (x <= y)                                       %(MaxYDef)%
 . (max x y == x) = (y <= x)                                       %(MaxXDef)%
 . (min x y == x) = (x <= y)                                       %(MinXDef)%
 . (min x y == y) = (y <= x)                                       %(MinYDef)%
 . (max x y == y) = (max y x == y)                        %(MaxSym)% %implied
 . (min x y == y) = (min y x == y)                        %(MinSym)% %implied
```

As declarações de instância dos tipos `Ordering` e `Bool`, para a classe `Ord`, seguiram o mesmo padrão das demais: definiu-se o funcionamento da função de ordenação total sobre os construtores de cada tipo; as demais funções foram escritas como teoremas. O tipo `Unit`, por possuir apenas um construtor, teve todas as funções declaradas como teoremas porque as mesmas podem ser provadas diretamente dos axiomas.

4.5 Especificações *Maybe*, *Either*, *MaybeMonad* e *Either-Functor*

Os tipos de dado `Maybe a` e `Either a b`, onde `a` e `b` são variáveis de tipo, foram desenvolvidos em duas fases.

No primeiro passo, o tipo de dado *Maybe* (ver Apêndice A.5, na página 96) foI especificado, acompanhado de uma função de mapeamento e declarações de instância para as classes `Eq` e `Ord`.

```
spec Maybe = Eq and Ord then
var a,b,c : Type;
    e : Eq;
    o : Ord;
free type Maybe a ::= Just a | Nothing
var x : a;
    y : b;
    ma : Maybe a;
    f : a -> b
...
type instance Maybe e: Eq
var x,y : e;
. (Just x == Just y) = True <=> (x == y) = True        %(IME01)%
...
. Just x == Nothing = False                            %(IME03)%
type instance Maybe o: Ord
var x,y : o;
. (Nothing < Just x) = True                            %(IMO01)%
. (Just x < Just y) = (x < y)                          %(IMO02)%
...
end
```

As declarações de classes foram feitas através da aplicação das funções das classes sobre elementos de um mesmo construtor, para os dois construtores existentes em cada tipo, e em seguida, entre dois elementos de construtores diferentes. As aplicações da função `__==__` entre construtores com variáveis de tipo e da função `__<__` foram inseridas como axiomas. A igualdade entre construtores sem variáveis de tipo e as demais funções de ordenação foram definidas como teoremas porque seu comportamento pode ser deduzido a partir dos axiomas anteriores.

Tratamento parecido foi utilizado na especificação do tipo de dado *Either* (ver Apêndice A.7, na página 98).

```
spec Either = Eq and Ord then
var a, b, c : Type; e, ee : Eq; o, oo : Ord;
free type Either a b ::= Left a | Right b
...
type instance Either e ee: Eq
var x,y : e; z,w : ee;
. ((Left x : Either e ee) ==
   (Left y : Either e ee)) = (x == y)                     %(IEE01)%
. ((Right z : Either e ee) ==
   (Right w : Either e ee)) = (z == w)                    %(IEE02)%
. ((Left x : Either e ee) ==
   (Right z : Either e ee)) = False                       %(IEE03)%
type instance Either o oo: Ord
var x,y : o; z,w : oo;
. ((Left x : Either o oo) < (Right z : Either o oo))
     = True                                               %(IEO01)%
. ((Left x : Either o oo) < (Left y : Either o oo))
     = (x < y)                                            %(IEO02)%
. ((Right z : Either o oo) < (Right w : Either o oo))
     = (z < w)                                            %(IEO03)%
```

```
...
end
```

Na segunda fase, as especificações foram estendidas para incluir declarações para classes envolvendo functores e mônadas. O tipo `Maybe a` (ver Apêndice A.6, na página 97) foi declarado como instância das classes `Functor` e `Monad`.

```
spec MaybeMonad = Maybe and Monad then
var a,b,c : Type;
         e : Eq;
         o : Ord;
type instance Maybe: Functor
vars  x: Maybe a;
      f: a -> b;
      g: b -> c
. map (\ y: a .! y) x = x                        %(IMF01)% %implied
. map (\ y: a .! g (f y)) x = map g (map f x)    %(IMF02)% %implied
type instance Maybe: Monad
vars  x, y: a;
      p: Maybe a;
      q: a ->? Maybe b;
      r: b ->? Maybe c;
      f: a ->? b
. def q x => ret x >>= q = q x                   %(IMM01)% %implied
. p >>= (\ x: a . ret (f x) >>= r)
     = p >>= \ x: a . r (f x)                    %(IMM02)% %implied
. p >>= ret = p                                  %(IMM03)% %implied
. (p >>= q) >>= r = p >>= \ x: a . q x >>= r     %(IMM04)% %implied
. (ret x : Maybe a) = ret y => x = y             %(IMM05)% %implied
var x : Maybe a;
    f : a -> b;
```

```
. map f x = x >>= (\ y:a . ret (f y))                  %(T01)% %implied
end
```

Já o tipo `Either a b` (ver Apêndice A.8, na página 100) foi declarado como instância da classe `Functor`.

```
spec EitherFunctor = Either and Functor then
var a, b, c : Type;
    e, ee : Eq;
    o, oo : Ord;
type instance Either a: Functor
vars x: Either c a;
     f: a -> b;
     g: b -> c
. map (\ y: a .! y) x = x                             %(IEF01)% %implied
. map (\ y: a .! g (f y)) x = map g (map f x)         %(IEF02)% %implied
end
```

Esta separação realizada foi necessária devido à falta de suporte, por parte da ferramenta Hets, para tradução de construtores de classe de ordens superiores de HasCasl para HOL. Ela também permitiu que as provas que não envolviam funtores e mônadas pudessem ser escritas, já que a inclusão das declarações de instância para classes com construtores de segunda ordem impediria que os demais axiomas pudessem ser provados.

4.6 Especificando funções através das especificações *Composition* e *Function*

Antes de especificar funções da biblioteca Prelude, fez-se necessário escolher entre importar ou recriar a especificação para composição de funções. Tendo em vista, novamente, que mudanças seriam necessárias quando a especificação fosse refinada para uso de avaliação preguiçosa, a mesma foi reescrita. Trocou-se o uso da λ-abstração pela aplicação direta da função porque a aplicação do axioma em questão nas provas geradas para o Isabelle torna-se mais simples. Em seguida, foram especificadas funções básicas como a função identidade, funções para operar sobre tuplas e funções para transformar funções *curried* em funções *uncurried* e vice-versa, como visto a seguir.

```
spec Composition =
vars a,b,c : Type
fun __o__ : (b -> c) * (a -> b) -> (a -> c);
vars a,b,c : Type; y:a; f : b -> c; g : a -> b
     . ((f o g) y) = f (g y)        %(Comp1)%
end

spec Function = Composition then
var a,b,c: Type; x: a; y: b; f: a -> b -> c; g: (a * b) -> c
fun id: a -> a
fun flip: (a -> b -> c) -> b -> a -> c
fun fst: (a * b) -> a
fun snd: (a * b) -> b
fun curry: ((a * b) -> c) -> a -> b -> c
fun uncurry: (a -> b -> c) -> (a * b) -> c
. id x = x                                         %(IdDef)%
. flip f y x = f x y                             %(FlipDef)%
. fst (x, y) = x                                  %(FstDef)%
. snd (x, y) = y                                  %(SndDef)%
. curry g x y = g (x, y)                        %(CurryDef)%
. uncurry f (x,y) = f x y                     %(UncurryDef)%
end
```

4.7 Utilizando especificações numéricas

Quando do início deste trabalho, não era possível a tradução de um código escrito em HasCasl para seu equivalente escrito em HOL quando o primeiro importasse especificações que utilizassem subtipos, como é o caso das especificações numéricas existentes na biblioteca de Casl. Reescrever todas as especificações estava fora do escopo do trabalho e geraria um retrabalho exaustivo e desnecessário. A solução inicial foi remover o uso de subtipos e criar funções de coerção de tipos. Posteriormente, como a ferramenta Hets passou a tratar corretamente o uso de subtipos, foi possível abandonar tal estratégia em prol da importação direta das bibliotecas preexistentes em Casl.

As especificações numéricas escritas em Casl, embora completas, geram especificações equivalentes em HOL que não conseguem ser manipuladas pelo provador Isabelle nos processos de reescrita. A manipulação destas especificações dependeria da criação de diversos lemas e teoremas auxiliares para que o provador Isabelle pudesse usar os axiomas destas especificações em processos de reescrita. Uma outra abordagem seria a crianção de isomorfismos mapeando os tipos e funções destas especificações para os respectivos tipos e funções dos tipos primitivos da linguagem HOL. Esta segunda abordagem, além de evitar a duplicação de tipos de dados com fins idênticos, ainda facilitaria as provas envolvendo tipos numéricos, uma vez que Isabelle possui vários métodos automáticos para lidar com estes tipos. Embora já exista um isomorfismo no repositório de códigos de Casl, não foi possível utilizá-lo para as provas por ter sido escrito para uma versão anterior da ferramenta HOL. Uma nova versão encontra-se em desenvolvimento.

O uso de especificações numéricas, devido ao exposto anteriormente, aumenta ainda mais a complexidade das provas escritas em HOL. Como o projeto prioriza as especificações ao invés das provas, e como o desenvolvimento de provas envolvendo números, dada a complexidade das provas, consumia muito tempo, optou-se por não finalizá-las, concentrando-se os esforços em aumentar a quantidade de especificações escritas.

As especificações numéricas de CASL foram importadas e agrupadas para que classes numéricas fossem criadas tal qual existem na biblioteca Prelude (ver Apêndice A.12, na página 109). O primeiro passo consistiu em declarar os tipos numéricos como instâncias das classes `Eq` e `Ord`, como é visto a seguir.

```
from Basic/Numbers get Nat, Int, Rat

spec NumericClasses = Ord and Nat and Int and Rat then
type instance Pos: Eq
type instance Pos: Ord
type instance Nat: Eq
type instance Nat: Ord
type instance Int: Eq
type instance Int: Ord
type instance Rat: Eq
type instance Rat: Ord
```

Criou-se uma classe `Num`, subclasse de `Eq`, com as operações esperadas para tipos numéricos. Um teorema envolvendo o valor absoluto e o sinal de elementos de tipos numéricos foi definido logo após a classe.

```
class Num < Eq {
 vars a: Num;
 x,y : a
 fun __+__: a * a -> a
 fun __*__: a * a -> a
 fun __-__: a * a -> a
 fun negate: a -> a
 fun abs: a -> a
 fun signum: a -> a
```

```
 fun fromInteger: Int -> a
}
vars a: Num;
     x,y : a
. (abs x) * (signum x) = x                            %(AbsSignumLaw)% %implied
```

Em seguida, para cada tipo numérico, efetuou-se a declaração de instância da classe `Num`, mapeando-se as operações da classe para as operações definidas nas especificações importadas, fazendo conversões de tipo, quando necessário, através da função de coerção de tipos `__as__`, como visto no caso do tipo `Nat`, a seguir.

```
type instance Nat: Num
vars a: Num;
     x,y: Nat;
     z: Int
. x + y = (__+__: Nat * Nat -> Nat) (x,y)                   %(INN01)%
. x * y = (__*__: Nat * Nat -> Nat) (x,y)                   %(INN02)%
. x - y = (__-!__: Nat * Nat -> Nat) (x,y)                  %(INN03)%
. negate x = 0 -! x                                         %(INN04)%
. (fun abs: a -> a) x = x                                   %(INN05)%
. signum x = 1                                              %(INN06)%
. fromInteger z = z as Nat                                  %(INN07)%
```

As classes `Integral`, vista a seguir, e a classe `Fractional` também foram definidas, embora a primeira ainda não seja subclasse das classes `Real` e `Enum`, que não foram incluídas na especificação pela complexidade envolvida na manipulação destas e do tipo inteiro de máquina, limitado pelo tamanho da palavra do processador.

```
class Integral < Num
{
vars a: Integral;
```

```
fun __quot__, __rem__, __div__, __mod__: a * a -> a
fun quotRem, divMod: a -> a -> (a * a)
fun toInteger: a -> Int
}

type instance Nat: Integral
type instance Int: Integral
type instance Rat: Integral

vars a: Integral;
     x,y,z,w,r,s: a;
. (z,w) = quotRem x y => x quot y = z                   %(IRI01)%
. (z,w) = quotRem x y => x rem y = w                    %(IRI02)%
. (z,w) = divMod x y => x div y = z                     %(IRI03)%
. (z,w) = divMod x y => x mod y = w                     %(IRI04)%
. signum w = negate (signum y) /\ (z,w) = quotRem x y
    => divMod x y =
        (z - (fromInteger (toInteger (1:Nat))) , w + s)     %(IRI05)%
. not (signum w = negate (signum y))
    /\ (z,w) = quotRem x y
        => divMod x y = (z, w)                          %(IRI06)%
```

4.8 Especificando listas e operações associadas

A especificação de listas foi a primeira a necessitar de tipos numéricos. Com base na discussão apresentada anteriormente, decidiu-se dividir a especificação em duas outras, separando-se o tipo de dado e as funções que não necessitavam de tipos numéricos das funções que envolviam estes tipos. Assim, os teoremas da primeira especificação poderiam ser provados como nas especificações anteriores, e os que envolviam números poderiam não ser provados, provocando um comprometimento mínimo da especificação quanto à sua validação.

A primeira especificação (ver Apêndice A.11, na página 101) foi dividida em seis partes, a fim de agrupar funções relacionadas da mesma forma como é feito na biblioteca Prelude. Primeiramente, definiu-se o tipo de dado `free type List a`, dependente do tipo `a`, com os construtores `Nil` e `Cons a (List a)`.

```
spec ListNoNumbers = Function and Ord then
var a : Type
free type List a ::= Nil | Cons a (List a)
```

Em seguida, definiram-se algumas funções básicas, onde pela primeira vez foi necessário o uso de funções parciais, uma vez que as funções `head` e `tail` não são definidas para listas vazias.

```
var a,b : Type
fun head : List a ->? a;
fun tail : List a ->? List a;
fun foldr : (a -> b -> b) -> b -> List a -> b;
fun foldl : (a -> b -> a) -> a -> List b -> a;
fun map : (a -> b) -> List a -> List b;
fun filter : (a -> Bool) -> List a -> List a;
fun __++__ : List a * List a -> List a;
fun zip : List a -> List b -> List (a * b);
fun unzip : List (a * b) -> (List a * List b)
```

Na segunda parte da especificação estão as declarações de instância para as classes `Eq` e `Ord`. As funções foram definidas de forma análoga às declarações de instância das especificações anteriormente descritas.

```
var a : Eq; x,y: a; xs, ys: List a
type instance List a: Eq
. ((Cons x xs) == (Cons y ys)) = ((x == y) && (xs == ys))        %(ILE02)%
var b : Ord; z,w: b; zs, ws: List b
type instance List b: Ord
. (z < w) = True => ((Cons z zs) < (Cons w ws)) = True           %(ILO05)%
. (z == w) = True => ((Cons z zs) < (Cons w ws)) = (zs < ws)     %(ILO06)%
. (z < w) = False /\ (z == w) = False
     => ((Cons z zs) < (Cons w ws)) = False                      %(ILO07)%
```

A terceira parte concentra quatro teoremas que relacionam entre si algumas funções da primeira parte da especificação. Embora estes teoremas definam como as funções interagem, eles não devem ser axiomas porque são consequência da definição das funções. O uso da diretiva `%implies` indica que todas as equações definidas nesta subparte são consideradas teoremas.

```
. foldl i e (ys ++ ts)
        = foldl i (foldl i e ys) ts       %(FoldlDecomp)%
. map f (xs ++ zs)
        = (map f xs) ++ (map f zs)        %(MapDecomp)%
. map (g o f) xs = map g (map f xs)   %(MapFunctor)%
. filter p (map f xs)
        = map f (filter (p o f) xs)       %(FilterProm)%
```

A quarta seção inclui algumas outras funções básicas, que complementam as funções da primeira seção, além de funções que mapeiam outras funções sobre elementos de uma lista. Novamente, algumas funções foram definidas como parciais por não serem definidas para listas vazias.

```
fun init: List a ->? List a;
fun last: List a ->? a;
```

```
fun null: List a -> Bool;
fun reverse: List a -> List a;
fun foldr1: (a -> a -> a) -> List a ->? a;
fun foldl1: (a -> a -> a) -> List a ->? a;
fun scanl: (a -> b -> a) -> a -> List b -> List a
fun scanl1: (a -> a -> a) -> List a -> List a
fun scanr: (a -> b -> b) -> b -> List a -> List b
fun scanr1: (a -> a -> a) -> List a -> List a
```

A quinta parte apresenta funções lógicas sobre listas e funções responsáveis pela criação de sub-listas, segundo um dado predicado.

```
fun andL : List Bool -> Bool;
fun orL : List Bool -> Bool;
fun any : (a -> Bool) -> List a -> Bool;
fun all : (a -> Bool) -> List a -> Bool;
fun concatMap : (a -> List b) -> List a -> List b;
fun concat : List (List a) -> List a;
fun maximum : List d ->? d;
fun minimum : List d ->? d;
fun takeWhile : (a -> Bool) -> List a -> List a
fun dropWhile : (a -> Bool) -> List a -> List a
fun span : (a -> Bool) -> List a -> (List a * List a)
fun break : (a -> Bool) -> List a -> (List a * List a)
```

A última seção da especificação apresenta funções sobre listas que não estão definidas na biblioteca Prelude, mas estão presentes em outras bibliotecas de Haskell e que foram necessárias para definir algumas especificações presentes neste trabalho.

```
fun insert: d -> List d -> List d
fun delete: e -> List e -> List e
fun select: (a -> Bool) -> a -> (List a * List a) -> (List a * List a)
fun partition: (a -> Bool) -> List a -> (List a * List a)
```

4.9 Agrupando listas e funções com tipos numéricos

A especificação de listas envolvendo tipos numéricos (ver Apêndice A.13, na página 113), como pode ser vista abaixo, inclui funções para tamanho de listas, criação de sub-listas de tamanho definido, divisão de lista em posição definida e soma e produto de todos os elementos de uma lista. Duas funções auxiliares, para soma e produto, foram criadas, de forma a facilitar a legibilidade da especificação. Como não deveriam ser utilizadas fora do escopo desta especificação, as funções foram tornadas invisíveis através da construção `hide sum', product'`. No entanto, quando a ferramenta HETS é utilizada para traduzir esta especificação, a mesma sugere que seja utilizada uma versão sem funções ocultas. Apesar do aviso, as especificações que importam a especificação com elementos ocultos são traduzidas normalmente pela ferramenta HETS.

```
spec ListWithNumbers = ListNoNumbers and NumericClasses then {
vars a,b: Type;
     c,d: Num;
     x,y : a;
     xs,ys : List a;
     n,nx : Int;
     z,w: Int;
     zs,ws: List Int
fun length: List a -> Int;
fun take: Int -> List a -> List a
fun drop: Int -> List a -> List a
fun splitAt: Int -> List a -> (List a * List a)
fun sum: List c -> c
fun sum': List c -> c -> c
fun product: List c -> c
fun product': List c -> c -> c
...
```

```
} hide sum', product'
end
```

4.10 Adicionando funções numéricas

Uma abordagem semelhante à empregada para listas foi utilizada para incluir funções que utilizam tipos numéricos (ver Apêndice A.14, na página 115). Funções de paridade, exponenciação, cálculo de mínimo múltiplo comum e de máximo divisor comum foram definidas estendendo-se a especificação de funções. Novamente, foram usadas funções temporárias, escondidas das demais especificações, como visto a seguir.

```
spec NumericFunctions = Function and NumericClasses then {
var a: Num;
    b: Integral;
    c: Fractional
fun subtract: a -> a -> a
fun even: b -> Bool
fun odd: b -> Bool
fun gcd: b -> b ->? b
fun lcm: b -> b -> b
fun gcd': b -> b -> b
fun __^__: a * b -> a
fun f: a -> b -> a
fun g: a -> b -> a -> a
...
} hide f,g
end
```

4.11 Suporte a caracteres e cadeias de caracteres

Para suportar caracteres e cadeias de caracteres, importou-se a especificação existente na biblioteca de CASL e adicionaram-se declarações de instância de classes para as classes `Eq` e `Ord`. A especificação `Char`, de CASL, foi renomeada ao ser importada para evitar conflitos de nomes. Esta especificação mapeia os códigos hexadecimais que representam caracteres no sistema *Unicode* para um elemento identificado por um número natural. Em seguida, os caracteres propriamente ditos são relacionados com os seus respectivos códigos hexadecimais. As declarações de classe definem o funcionamento das funções `__==__` e `__<__` sobre o construtor do tipo de dado `Char`. As demais funções, que decorrem das duas definições anteriores,foram marcadas como teoremas, como se vê abaixo.

```
from Basic/CharactersAndStrings get Char |-> IChar


spec Char = IChar and Ord and NumericClasses then
vars x, y: Char
type instance Char: Eq
. (ord(x) == ord(y)) = (x == y)                              %(ICE01)%
. Not(ord(x) == ord(y)) = (x /= y)                  %(ICE02)% %implied
type instance Char: Ord
. (ord(x) < ord(y)) = (x < y)                                %(ICO04)%
. (ord(x) <= ord(y)) = (x <= y)                     %(ICO05)% %implied
. (ord(x) > ord(y)) = (x > y)                       %(ICO06)% %implied
. (ord(x) >= ord(y)) = (x >= y)                     %(ICO07)% %implied
. (compare x y == EQ) = (ord(x) == ord(y))          %(ICO01)% %implied
. (compare x y == LT) = (ord(x) < ord(y))           %(ICO02)% %implied
. (compare x y == GT) = (ord(x) > ord(y))           %(ICO03)% %implied
. (ord(x) <= ord(y)) = (max x y == y)               %(ICO08)% %implied
. (ord(y) <= ord(x)) = (max x y == x)               %(ICO09)% %implied
. (ord(x) <= ord(y)) = (min x y == x)               %(ICO10)% %implied
```

```
. (ord(y) <= ord(x)) = (min x y == y)                    %(ICO11)% %implied
end
```

A especificação de cadeias de caracteres define o tipo de dado `String` como um apelido para o tipo `List Char` através do operador `:=`. Com esta definição, não foi criado um novo tipo de dado, mas um outro nome, mais simples, para se referenciar o tipo composto. Dessa forma, as operações definidas para o tipo `List` são válidas para o tipo `String` e nenhuma definição adicional é necessária para que o tipo esteja bem definido. Alguns teoremas simples foram criados para verificar se o comportamento deste tipo era o esperado.

```
spec String = %mono
     ListNoNumbers and Char then
type String := List Char
vars a,b: String; x,y,z: Char; xs, ys: String
. x == y = True =>
         ((Cons x xs) == (Cons y xs)) = True                 %(StringT1)% %implied
. xs /= ys = True =>
        ((Cons x ys) == (Cons y xs)) = False                 %(StringT2)% %implied
. (a /= b) = True =>  (a == b) = False                   %(StringT3)% %implied
. (x < y) = True =>
        ((Cons x xs) < (Cons y xs)) = True                   %(StringT4)% %implied
. (x < y) = True /\ (y < z) = True => ((Cons x (Cons z Nil))
         < (Cons x (Cons y Nil))) = False                %(StringT5)% %implied
end
```

4.12 Definindo listas de tipos monádicos

Listas de tipos monádicos, muito utilizadas no sequenciamento de ações de entrada e saída em programas Haskell, foram especificadas a partir das listas sem tipos numéricos.

Funções para sequências de ações e mapeamento de funções monádicas sobre listas foram definidas tal qual na biblioteca Prelude, como visto abaixo.

```
spec MonadicList = Monad and ListNoNumbers then
vars a,b: Type;
     m: Monad;
     f: a -> m b;
     ms: List (m a);
     k: m a -> m (List a) -> m (List a);
     n: m a;
     nn: m (List a);
     x: a;
     xs: List a;
fun sequence: List (m a) -> m (List a)
fun sequenceUnit: List (m a) -> m Unit
fun mapM: (a -> m b) -> List a -> m (List b)
fun mapMUnit: (a -> m b) -> List a -> m (List Unit)
. sequence ms = let
  k n nn = n >>= \ x:a. (nn >>= \ xs: List a . (ret (Cons x xs))) in
    foldr k (ret (Nil: List a)) ms                      %(SequenceListDef)%
end
```

4.13 Exemplificando o uso da biblioteca desenvolvida

A fim de exemplificar o uso da biblioteca desenvolvida, foram criadas duas especificações envolvendo algoritmos de ordenação. Foram escolhidas funções de ordenação porque envolveriam listas e poderiam não envolver números. Desta forma, poder-se-ia tentar provar os teoremas, completando os exemplos.

A primeira especificação (ver Apêndice A.19, na página 120), mais simples, utilizou os algoritmos *Insertion Sort* e *Quick Sort*. As funções de ordenação foram definidas através de funções da biblioteca criada e de λ-abstrações totais, usadas como parâmetros para as aplicações da função `filter`. Para verificar a correção da especificação, foram criados quatro teoremas que aplicam as funções de ordenação, como visto abaixo.

```
spec ExamplePrograms = ListNoNumbers then
var a: Ord;
    x,y: a;
    xs,ys: List a
fun quickSort: List a -> List a
fun insertionSort: List a -> List a
. quickSort (Nil: List a) = Nil                                 %(QuickSortNil)%
. quickSort (Cons x xs)
    = ((quickSort (filter (\ y:a .! y < x) xs))
      ++ (Cons x Nil))
        ++ (quickSort (filter (\ y:a .! y >= x) xs))          %(QuickSortCons)%
. insertionSort (Nil: List a) = Nil                         %(InsertionSortNil)%
. insertionSort (Cons x xs)
    = insert x (insertionSort xs)                      %(InsertionSortConsCons)%
then %implies
var a: Ord;
    x,y: a;
    xs,ys: List a
. andL (Cons True (Cons True (Cons True Nil))) = True        %(Program01)%
. quickSort (Cons True (Cons False (Nil: List Bool)))
     = Cons False (Cons True Nil)                            %(Program02)%
. insertionSort (Cons True (Cons False (Nil: List Bool)))
     = Cons False (Cons True Nil)                            %(Program03)%
. insertionSort xs = quickSort xs                            %(Program04)%
end
```

Na segunda especificação (ver Apêndice A.19, na página 120), um novo tipo de dado foi criado (`Split a b`) para ser usado como representação interna nas funções de ordenação. A ideia utilizada na especificação foi dividir uma lista e, em seguida, unir as suas partes de acordo com o algoritmo escolhido.

```
spec SortingPrograms = ListWithNumbers then
var a,b : Ord;
free type Split a b ::= Split b (List (List a))
```

Em seguida, definiu-se uma função de ordenação genérica, chamada `GenSort`, que aplica as funções de divisão e união sobre uma lista.

```
fun genSort:(List a -> Split a b)-> (Split a b -> List a)-> List a -> List a
...
. xs = (Cons x (Cons y ys)) /\ split xs = Split r xxs
     => genSort split join xs
          = join (Split r (map (genSort split join) xxs))        %(GenSortT1)%
. xs = (Cons x (Cons y Nil)) /\ split xs = Split r xxs
     => genSort split join xs
          = join (Split r (map (genSort split join) xxs))        %(GenSortT2)%
. xs = (Cons x Nil) \/ xs = Nil
     => genSort split join xs = xs                                %(GenSortF)%
. splitInsertionSort (Cons x xs)
```

O algoritmo `Insertion Sort` foi implementado com o auxílio de uma função responsável por inserir elementos em uma lista através da função `insert`.

```
fun splitInsertionSort: List b -> Split b b
fun joinInsertionSort: Split a a -> List a
fun insertionSort: List a -> List a
...
     = Split x (Cons xs (Nil: List (List a)))          %(SplitInsertionSort)%
. joinInsertionSort (Split x (Cons xs (Nil: List (List a))))
```

```
     = insert x xs                                           %(JoinInsertionSort)%
. insertionSort xs
     = genSort splitInsertionSort joinInsertionSort xs    %(InsertionSort)%
```

O algoritmo `Quick Sort` utiliza uma função de divisão que particiona a lista em duas novas listas com o uso da função `__<__`, passada como parâmetro através de uma λ-abstração total.

```
fun splitQuickSort: List a -> Split a a
fun joinQuickSort: Split b b -> List b
fun quickSort: List a -> List a
...
. splitQuickSort (Cons x xs)
     = let (ys, zs) = partition (\t:a .! x < t) xs
       in Split x (Cons ys (Cons zs Nil))                  %(SplitQuickSort)%
. joinQuickSort (Split x (Cons ys (Cons zs Nil)))
     = ys ++ (Cons x zs)                                    %(JoinQuickSort)%
. quickSort xs = genSort splitQuickSort joinQuickSort xs     %(QuickSort)%
```

O algoritmo `Selection Sort` faz uso de uma função de divisão que depende da função `minimum` para extrair o menor elemento de uma lista.

```
fun splitSelectionSort: List a -> Split a a
fun joinSelectionSort: Split b b -> List b
fun selectionSort: List a -> List a
...
. splitSelectionSort xs = let x = minimum xs in
        Split x (Cons (delete x xs) (Nil: List(List a)))    %(SplitSelectionSort)%
. joinSelectionSort (Split x (Cons xs Nil)) =
         (Cons x xs)                                          %(JoinSelectionSort)%
. selectionSort xs
     = genSort splitSelectionSort joinSelectionSort xs   %(SelectionSort)%
```

O algoritmo `Merge Sort` divide uma lista ao meio e, então, une as duas listas, ordenando-as recursivamente.

```
fun splitMergeSort: List b -> Split b Unit
fun joinMergeSort: Split a Unit -> List a
fun merge: List a -> List a -> List a
fun mergeSort: List a -> List a
...
. def((length xs) div 2) /\ n = ((length xs) div 2)
     => splitMergeSort xs = let (ys,zs) = splitAt n xs
        in Split () (Cons ys (Cons zs Nil))                %(SplitMergeSort)%
. xs = (Nil: List a) => merge xs ys = ys                         %(MergeNil)%
. xs = (Cons v vs) /\ ys = (Nil: List a)
     => merge xs ys = xs                                     %(MergeConsNil)%
. xs = (Cons v vs) /\ ys = (Cons w ws) /\ (v < w) = True
     => merge xs ys = Cons v (merge vs ys)                 %(MergeConsConsT)%
. xs = (Cons v vs) /\ ys = (Cons w ws) /\ (v < w) = False
     => merge xs ys = Cons w (merge xs ws)                 %(MergeConsConsF)%
. joinMergeSort (Split () (Cons ys (Cons zs Nil)))
     = merge ys zs                                          %(JoinMergeSort)%
. mergeSort xs = genSort splitMergeSort joinMergeSort xs    %(MergeSort)%
```

A fim de verificar propriedades acerca das definições das funções, foram definidos três predicados. O primeiro, `__elem__`, verifica se um elemento pertence a uma lista; o predicado `isOrdereded` garante que uma lista está ordenada de forma correta; por fim, o predicado `permutation` verifica se uma lista é uma permutação de uma segunda lista, ou seja, verifica se ambas as listas possuem os mesmos elementos. Embora o predicado `%(PermutationCons)%` fosse desnecessário para definir o predicado `permutation`, o primeiro se mostrou necessário nas provas envolvendo este último.

```
vars a: Ord;
```

```
     x,y: a;
     xs,ys: List a
preds __elem__ : a * List a;
      isOrdered: List a;
      permutation: List a * List a
. not x elem (Nil: List a)                                           %(ElemNil)%
. x elem (Cons y ys) <=> x = y \/ x elem ys                          %(ElemCons)%
. isOrdered (Nil: List a)                                            %(IsOrderedNil)%
. isOrdered (Cons x (Nil: List a))                                   %(IsOrderedCons)%
. isOrdered (Cons x (Cons y ys))
     <=> (x <= y) = True /\ isOrdered(Cons y ys)       %(IsOrderedConsCons)%
. permutation ((Nil: List a), Nil)                                   %(PermutationNil)%
. permutation (Cons x (Nil: List a),
        Cons y (Nil: List a)) <=> x=y                                   %(PermutationCons)%
. permutation (Cons x xs, Cons y ys) <=>
     (x=y /\ permutation (xs, ys)) \/ (x elem ys
          /\ permutation(xs, Cons y (delete x ys)))   %(PermutationConsCons)%
```

Foram criados teoremas para garantir que a aplicação dos algoritmos sobre uma mesma lista, dois a dois, obtivessem a mesma lista como resultado; para verificar que a aplicação de cada algoritmo em uma lista resultasse em uma lista ordenada; e para provar que uma lista de entrada seja permutação da lista resultante da aplicação de cada função de ordenação sobre a lista inicial.

```
then %implies
var a,b : Ord;
    xs, ys : List a;
. insertionSort xs = quickSort xs                                    %(Theorem01)%
. insertionSort xs = mergeSort xs                                    %(Theorem02)%
. insertionSort xs = selectionSort xs                                %(Theorem03)%
. quickSort xs = mergeSort xs                                        %(Theorem04)%
```

```
. quickSort xs = selectionSort xs                                  %(Theorem05)%
. mergeSort xs = selectionSort xs                                  %(Theorem06)%
. isOrdered(insertionSort xs)                                      %(Theorem07)%
. isOrdered(quickSort xs)                                          %(Theorem08)%
. isOrdered(mergeSort xs)                                          %(Theorem09)%
. isOrdered(selectionSort xs)                                      %(Theorem10)%
. permutation(xs, insertionSort xs)                                %(Theorem11)%
. permutation(xs, quickSort xs)                                    %(Theorem12)%
. permutation(xs, mergeSort xs)                                    %(Theorem13)%
. permutation(xs, selectionSort xs)                                %(Theorem14)%
end
```

Capítulo 5

Ferramentas Hets e Isabelle

Ambas as ferramentas Hets e Isabelle possuem integração com o editor de texto *emacs*. Através dessa integração, pode-se escrever as especificações com coloração de sintaxe no editor e, então, utilizar as duas ferramentas para efetuar análise sintática e escrever provas. Neste capítulo, descreve-se sucintamente o uso de cada uma das ferramentas.

5.1 Verificando especificações com Hets

A ferramenta Hets pode ser invocada dentro do editor de texto *emacs* de duas formas: na primeira, ela apenas realiza a análise sintática da especificação contida na janela atual; na segunda forma, após realizar a análise sintática, o grafo de desenvolvimento é gerado, mostrando a estrutura da especificação analisada. Para a especificação da biblioteca deste trabalho, o grafo resultante pode ser visto na Figura 5.1, na página 66.

Na figura, os nós elípticos representam uma especificação do arquivo que foi analisado. Os círculos indicam subespecificações, ou seja, trechos de especificações que foram separados pela declaração `then`. Os retângulos indicam especificações importadas de outras bibliotecas. No caso deste trabalho, todas foram importadas da biblioteca da linguagem

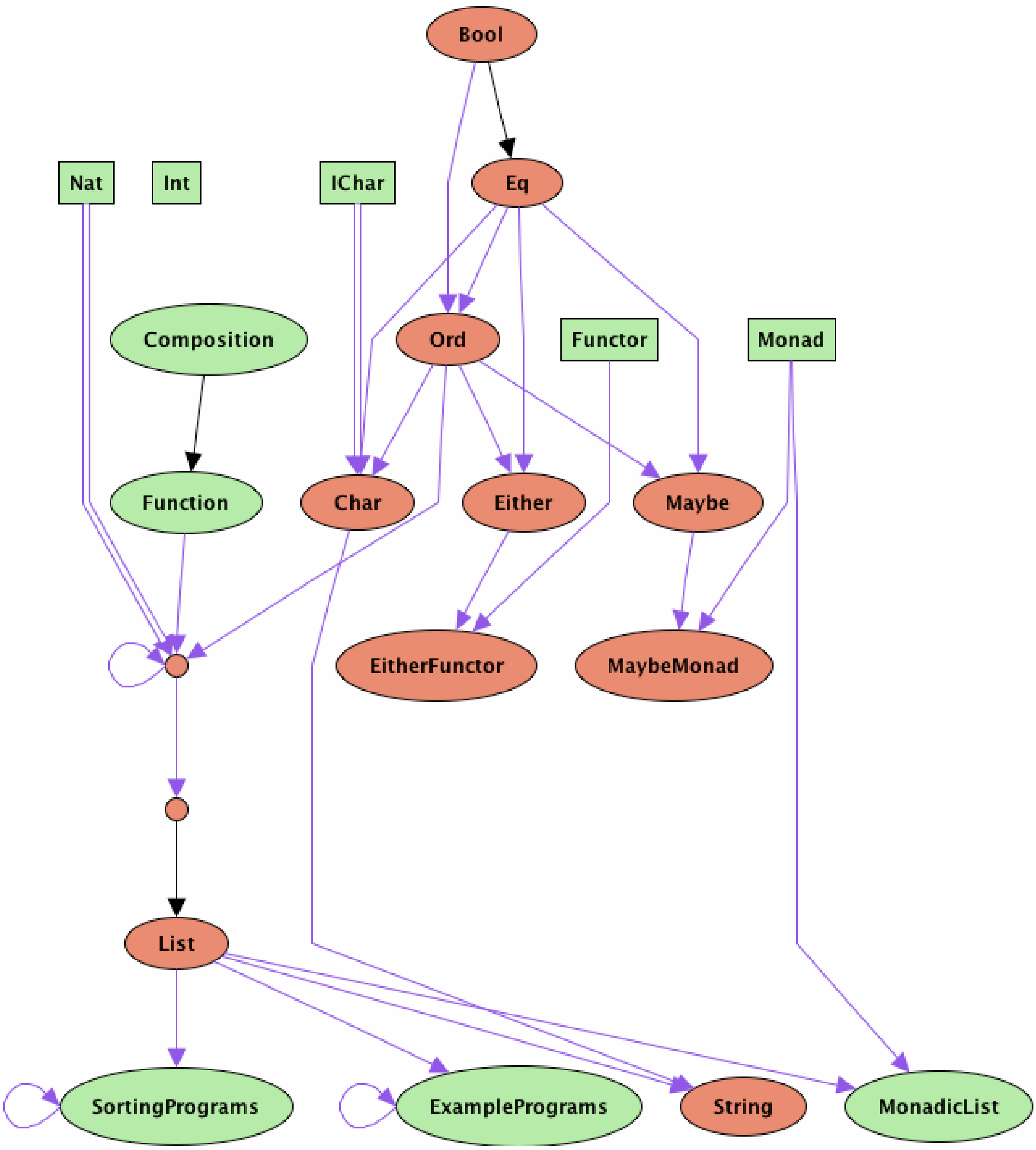

Figura 5.1: Inicio

CASL. Os nós vermelhos (cinza escuro) indicam especificações que possuem um ou mais teoremas a serem provados. Os nós verdes (cinza claro) não possuem teoremas ou todos os seus teoremas já estão provados.

A partir do grafo pode-se iniciar o provador de teoremas Isabelle para verificar os teoremas existentes nos nós. Uma prova típica se inicia pelo método automático de prova sobre a estrutura da especificação. Este método analisa as teorias presentes e as diretivas (`%mono`, `%implies`, etc.), construindo as dependências entre as especificações e revelando os nós ocultos das subespecificações que contêm teoremas a serem provados.

O próximo passo é utilizar o provador Isabelle para verificar os teoremas existentes nos nós vermelhos. Para tanto, basta escolher um nó e, com o botão direito, escolher a opção *Prove*. A janela seguinte permite escolher um entre vários provadores de teoremas. Para a linguagem HASCASL, o provador Isabelle precisa ser usado. Esta opção irá efetuar a tradução da especificação para uma teoria correspondente em HOL, exibindo a mesma em uma outra janela do editor *emacs*, permitindo com que a prova dos teoremas seja escrita.

Após processar todo o arquivo de provas, pode-se fechar o editor de texto e o estado desta prova será atualizado no grafo. Se a prova foi finalizada com sucesso, o nó correspondente terá sua cor alterada para verde. Caso contrário, sua cor permanecerá vermelha. Quando todos os nós circulares correspondentes a subespecificações de um nó elíptico tiverem a cor alterada para verde, eles novamente se tornarão ocultos, e apenas o nó elíptico correspondente à especificação completa será exibido.

Algumas provas de teoremas da especificação da biblioteca ainda permanecem em aberto, como ilustram os nós vermelhos da Figura 5.2, na página 68. A maior parte das provas em aberto estão relacionadas à falta de suporte de construtores de classes na linguagem HOL. Métodos alternativos de tradução para Isabelle/HOL estão em investigação pelo grupo responsável pela ferramenta HETS.

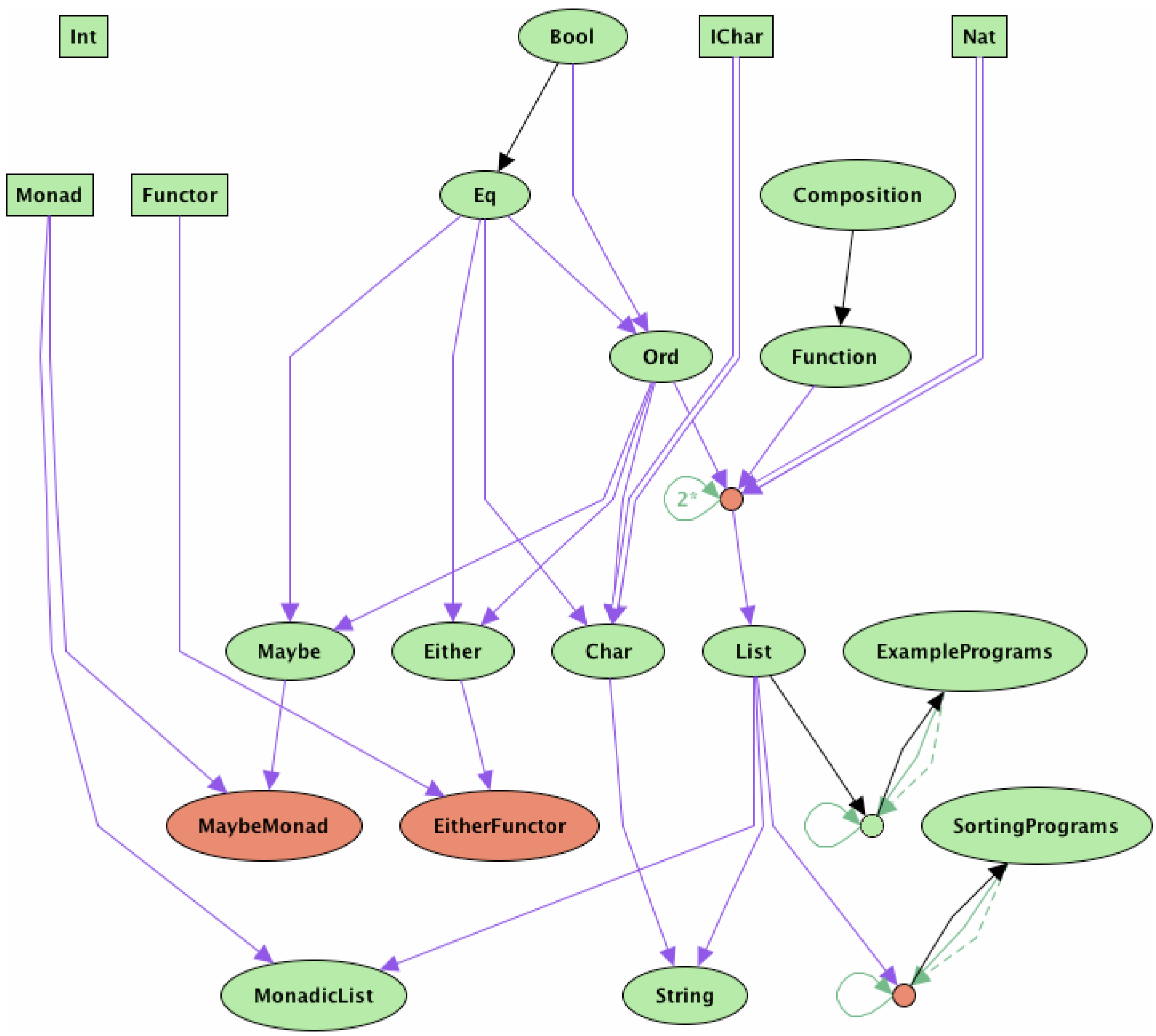

Figura 5.2: Estado atual.

5.2 Usando o provador de teoremas Isabelle

A tarefa de verificar os teoremas gerados pela especificação, embora não fosse o foco do trabalho, tornou-se a parte mais complexa. Embora ainda permaneçam teoremas em aberto, a maioria pode ser verificado. A seguir, explica-se como se deu o processo de construção das provas.

A maior parte das provas foi iniciada pelo comando `apply(auto)` porque desejava-se que Isabelle agisse de forma automática, sempre que possível. Abaixo, ilustra-se a prova de um teorema da especificação `Bool`:

```
theorem NotFalse1 : "ALL x.
 Not' x = True' = (x = False')"
apply(auto)
apply(case_tac x)
apply(auto)
done
```

A seguir, os comandos da prova são explicados:

- *apply (auto)*:

  Este comando simplifica a proposição atual de forma automática, indo o mais longe que conseguir nas reduções. Neste teorema, o comando apenas conseguiu eliminar o quantificador universal, produzindo o resultado a seguir:

  ```
  goal (1 subgoal):
   1. !!x. Not' x = True' ==> x = False'
  ```

- *apply (case_tac x)*:

  O método *case_tac* atribui uma valoração possível para a variável `x`, substituindo-a por cada um dos construtores do tipo a que a variável pertence. Aqui, como a variável `x` é do tipo `Bool`, `x` foi instanciado para os construtores `True` e `False`:

```
goal (2 subgoals):
 1. !!x. [| Not' x = True'; x = False' |] ==> x = False'
 2. !!x. [| Not' x = True'; x = True' |] ==> x = False'
```

- *apply (auto)*:

  Desta vez, o comando automático foi capaz de terminar a prova automaticamente.

  ```
  goal:
  No subgoals!
  ```

Um exemplo de prova para um teorema da especificação `Eq` é mostrado a seguir. Nesta prova, foi introduzido o comando `simp add:`, que espera uma lista de axiomas ou teoremas previamente provados como parâmetros para serem utilizados em uma tentativa automática de simplificar a proposição corrente. Este comando faz uso, além dos teoremas passados como parâmetro, dos demais axiomas no escopo da teoria atual. Se a proposição não pode ser reduzida, o comando produz um erro; caso contrário, uma nova proposição é gerada, caso a prova não tenha sido concluída automaticamente.

```
theorem DiffTDef :
"ALL x. ALL y. x /= y = True'
= (Not' (x ==' y) = True')"
apply(auto)
apply(simp add: DiffDef)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: DiffDef)
done
```

Os teoremas usados na prova anterior também foram utilizados em várias provas da especificação `Ord`. A prova do teorema `%(LeTAssimetry)%` destaca-se das demais por introduzir a necessidade de criar lemas auxiliares. Em alguns casos, um axioma ou teorema precisa ser reescrito de forma que o Isabelle consiga utilizá-lo em suas provas automáticas. Para tanto, cria-se um lema, que apesar do nome, funciona da mesma forma que um teorema. No caso do teorema `%(LeTAssimetry)%`, utilizou-se o comando `rule ccontr` para iniciar uma prova por contradição. Após algumas simplificações, o provador Isabelle não foi capaz de utilizar o axioma `%(LeIrreflexivity)%` para simplificar o objetivo e produziu:

```
goal (1 subgoal):
 1. !!x y. [| x <' y = True'; y <' x = True' |] ==> False
```

Foi necessário criar o lema auxiliar `LeIrreflContra`, provado automaticamente pelo provador Isabelle. O teorema foi interpretado internamente da seguinte forma:

```
?x <' ?x = True' ==> False
```

Pode-se, então, induzir a ferramenta Isabelle a usar o lema anterior forçando a atribuição do valor `x` para a variável `?x` através do comando `rule_tac x="x" in LeIrreflContra`. A mesma tática foi utilizada para forçar o uso do axioma `%(LeTTransitive)%`. A prova foi finalizada com o comando `by auto`.

```
lemma LeIrreflContra :
  " x <' x = True' ==> False"
by auto

theorem LeTAsymmetry :
"ALL x. ALL y. x <' y = True'
  --> y <' x = False'"
apply(auto)
apply(rule ccontr)
```

```
apply(simp add: notNot2 NotTrue1)
apply(rule_tac x="x" in LeIrreflContra)
apply(rule_tac y="y" in LeTTransitive)
by auto
```

Alguns usos do comando `apply(auto)` podem entrar em laços infinitos. Um exemplo ocorreu quando os teoremas das especificações `Maybe` e `Either` foram provados. Para evitar o laço infinito, a regra de eliminação do quantificador universal foi aplicada diretamente, usando o comando `apply(rule allI)`. O comando `rule` aplica o teorema ou axioma especificado diretamente, sem usar outras regras na redução. Para remover mais de um quantificador, pode-se incluir o sinal `+` após a regra, indicando que a mesma deve ser utilizada repetidamente, até que não seja possível nenhuma outra simplificação.

Após remover os quantificadores, o comando `simp only:` foi aplicado. Este comando, ao contrário do comando `simp add:`, utiliza apenas as regras passadas como parâmetros para tentar simplificar o objetivo atual. Na maior parte das vezes, os dois comandos podem ser usados sem distinção. Algumas vezes, no entanto, o comando `simp add:` pode entrar em laços infinitos; nestes casos, o comando `simp only:` deve ser usado para que a simplificação seja possível. Abaixo, mostra-se um teorema e sua respectiva prova para exemplificar o procedimento descrito.

```
theorem IMO03 : "ALL x. Nothing >=' Just(x) = False'"
apply(rule allI)
apply(simp only: GeqDef)
apply(simp only: GeDef OrDef)
apply(case_tac "Just(x) <' Nothing")
apply(auto)
done
```

A especificação `ListNoNumbers` possui apenas o teorema `FoldlDecomp` em aberto. Já a especificação `ListWithNumbers`, que possui quatro teoremas, tem todos os seus teoremas em aberto. No primeiro caso, quase todos os teoremas necessitaram de comandos de indução para serem provados. Isabelle executa indução sobre uma variável através do comando `induct_tac`. Este comando espera como parâmetro uma variável ou uma expressão sobre a qual deve executar o processo de indução. A seguir, é apresentado um exemplo envolvendo indução oriundo da especificação `ListNoNumbers`.

```
theorem FilterProm :
"ALL f. ALL p. ALL xs.
 X_filter p (X_map f xs) =
   X_map f (X_filter
     (X__o__X (p, f)) xs)"
apply(auto)
apply(induct_tac xs)
apply(auto)
apply(case_tac "p(f a)")
apply(auto)
apply(simp add: MapCons)
apply(simp add: FilterConsT)
apply(simp add: MapCons)
apply(simp add: FilterConsT)
done
```

As especificações `Char` e `String` usam combinações de comandos apresentados nos exemplos acima e, dessa forma, não são exemplificadas aqui.

As provas da especificação do exemplo `ExamplePrograms` embora longas, usaram apenas três comandos, basicamente: `simp only:`, `case_tac`, e `simp add:`. Como o comando `simp add:` consegue, geralmente, simplificações maiores, optou-se por tentar usá-lo sempre que possível. Nos casos em que o comando entrou em laços infinitos, ele foi trocado pelo comando `simp only:`. Um teorema nesta especificação ainda permanece em aberto. A seguir, a prova do teorema `Program02` é mostrada como exemplo.

```
theorem Program02 :
"quickSort(X_Cons True' (X_Cons False' Nil')) =
 X_Cons False' (X_Cons True' Nil')"
apply(simp only: QuickSortCons)
apply(case_tac "(%y. y <' True') False'")
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusNil)
apply(simp only: XPlusXPlusCons)
apply(simp only: XPlusXPlusNil)
apply(case_tac "(%y. y >=' True') False'")
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp add: LeFGeTEqTRel)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortCons)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusNil)
apply(simp only: XPlusXPlusCons)
apply(simp only: XPlusXPlusNil)
apply(simp only: IBO5)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortCons)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusNil)
apply(simp only: XPlusXPlusCons)
apply(simp only: XPlusXPlusNil)
apply(case_tac "(%y. y >=' True') False'")
apply(simp only: FilterNil FilterConsT FilterConsF)
```

```
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusCons)
apply(simp only: XPlusXPlusNil)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortCons)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusNil)
apply(simp only: XPlusXPlusCons)
apply(simp only: XPlusXPlusNil)
apply(simp add: LeFGeTEqTRel)
done
```

Todos os teoremas da especificação `SortingPrograms` ainda não tiveram suas provas finalizadas. Embora, para todos eles, várias proposições intermediárias tenham sido provadas, o caso geral ainda permanece em aberto. Para mostrar o progresso feito nas provas, um exemplo com alguns comentários é apresentado a seguir. O comando `prefer` é utilizado para escolher qual proposição se quer provar ao se utilizar o provador Isabelle no modo interativo. O comando `oops` indica ao provador para desistir da prova e seguir em frente no arquivo de provas.

```
theorem Theorem07 : "ALL xs. isOrdered(insertionSort(xs))"
apply(auto)
apply(case_tac xs)
(* Proof for xs=Nil *)
prefer 2
apply(simp only: InsertionSort)
apply(simp add: GenSortF)
(* Proof for general case *)
apply(simp only: InsertionSort)
```

```
apply(case_tac List)
apply(auto)
apply(case_tac "X_splitInsertionSort (X_Cons a (X_Cons aa Lista))")
(* Proof for xs= Cons a Nil *)
prefer 2
apply(simp add: GenSortF)
(* Proof for xs=Cons a as*)
apply(case_tac Lista)
apply(auto)
prefer 2
(* Proof for xs = Cons a (Cons b Nil)*)
oops
```

Capítulo 6

Referencial Teórico

Capítulo 7

Uma tentativa de extensão para inclusão de avaliação preguiçosa, recursão e tipos infinitos

A avaliação preguiçosa pode ser incluída em especificações escritas em HasCasl de duas formas. Em uma abordagem mais simples, pode-se apenas utilizar funções com avaliação preguiçosa sem que o comportamento das mesmas possa ser mapeado para o comportamento de funções na linguagem Haskell. A abordagem mais complexa, por sua vez, exige a alteração do domínio semântico das funções em HasCasl para um subconjunto de tipos onde a noção de especificação executável pode ser definida, permitindo recursão e não-terminação de programas. A seguir, discutimos as duas abordagens e a aplicação das mesmas no projeto.

7.1 Tipos com avaliação preguiçosa

A inclusão de tipos com avaliação preguiçosa, inicialmente, pode ser feita trocando-se os tipos nos perfis das funções. Todo tipo `T` deve ser substituído pelo tipo `?T`, que é a abreviação sintática para a função parcial `Unit ->? T`. Quando o tipo a ser alterado for a imagem de uma função parcial, é necessário alterar a função para uma função total, ou seja, uma função parcial com perfil `T ->? S`, ao incorporar tipos com avaliação preguiçosa passa a ser a função total `?T -> ?S`. Vale notar que a função total `T1 -> T2 ->` .. -> ?S., com imagem no tipo com avaliação preguiçaosa `?S`, equivale à função parcial `T1 -> T2 ->` .. ->? S, com imagem no tipo com avaliação estrita `S`.

Com o uso de funções com avaliação preguiçosa, o tipo de dado `Bool` poderia ser mapeado para o tipo `?Bool` ao invés de ser definido na especificação. Como os operadores lógicos de HasCasl são definidos sobre o tipo `?Unit`, este mapeamento permitiria que as aplicações de funções com tipo de retorno `Bool` fossem interpretadas como predicados. Desta forma, não seria necessário comparar uma aplicação de função com um dos construtores do tipo `Bool` para formar um termo. Seria possível, também, utilizar todas as funções lógicas de HasCasl diretamente sobre os termos formados pela aplicação de tais funções.

O tipo de dados `Bool`, por exemplo, deveria ser alterado de:

```
spec Bool = %mono
free type Bool ::= True | False
fun Not__ : Bool -> Bool
fun __&&__ : Bool * Bool -> Bool
fun __||__ : Bool * Bool -> Bool
fun otherwiseH: Bool
vars x,y: Bool
. Not(False) = True                 %(NotFalse)%
```

```
. Not(True) = False               %(NotTrue)%
. False && x = False              %(AndFalse)%
. True && x = x                   %(AndTrue)%
. x && y = y && x                 %(AndSym)%
. x || y = Not(Not(x) && Not(y))  %(OrDef)%
. otherwiseH = True               %(OtherwiseDef)%
. Not x = True <=> x = False      %(NotFalse1)% %implied
. Not x = False <=> x = True      %(NotTrue1)% %implied
. not (x = True) <=> Not x = True   %(notNot1)% %implied
. not (x = False) <=> Not x = False %(notNot2)% %implied
end
```

para:

```
spec Bool = %mono
type Bool := ?Unit
fun Not__ : ?Bool -> ?Bool
fun __&&__ : ?Bool * ?Bool -> ?Bool
fun __||__ : ?Bool * ?Bool -> ?Bool
fun otherwiseH: ?Bool
vars x,y: ?Bool
. Not(false)                      %(NotFalse)%
. not Not(true)                   %(NotTrue)%
. not (false && x)                %(AndFalse)%
. true && x = x                   %(AndTrue)%
. x && y = y && x                 %(AndSym)%
. x || y = Not(Not(x) && Not(y))  %(OrDef)%
. otherwiseH                      %(OtherwiseDef)%
. (Not x) = not x         %(NotFalse1)% %implied
. not (Not x) = x         %(NotTrue1)% %implied
. (not x) = Not x         %(notNot1)% %implied
```

```
. (not not x) = not Not x   %(notNot2)% %implied
end
```

Estas alterações melhorariam a legibilidade das especificações e simplificariam o processo de provas em Isabelle uma vez que o tipo de dado `?Unit` é mapeado para o tipo de dado `bool` de HOL. A aplicação das alterações na biblioteca pode ser conferida no Apêndice C, na página 149, e as respectivas provas, no Apêndice D, na página 184.

A falta de exemplos na literatura deixou algumas dúvidas com relação a aplicação da avaliação preguiçosa. A aplicação deste tipo de avaliação no tipo `List List a`, por exemplo, poderia gerar o tipo `?List List a` ou o tipo `?List(?List(?a))`. Nenhum destes tipos gera erro sintático na ferramenta Hets no perfil de funções. O uso do primeiro tipo, no entanto, não permitiu que as especificações fossem traduzidas para HOL porque a ferramenta Hets não era capaz de casar os tipos de retorno, de parâmetros de funções e de variáveis que fizessem uso desse tipo. O segundo tipo, por sua vez, permitiu a tradução das especificações para HOL.

Durante as provas, entretanto, os tipos de algumas funções não conseguiram ser casados e as provas não puderam avançar. Notadamente, não foi possível casar os tipos das funções definidas nas especificações com o tipo das funções definidas internamente pela ferramenta Hets e utilizadas de forma auxiliar durante a tradução de tipos com avaliação preguiçosa de HasCasl para HOL. Dessa forma, apenas as especificações `Bool`, `Eq`, `Ord`, `Maybe` e `Either` tiveram todos os seus teoremas provados. A especificação `ListNoNumbers` possui apenas um teorema em aberto. As demais especificações ou possuem a maior parte dos teoremas em aberto ou não puderam ter as provas iniciadas pelo problema descrito anteriormente.

A aplicação de tipos com avaliação preguiçosa definidos da forma descrita anteriormente não simula o comportamento da linguagem de programação Haskell, impedindo o uso de recursão e de estruturas de dados infinitas. Esta limitação, talvez, possa estar relacionada com o problema enfrentado na aplicação de avaliação preguiçosa no tipo `?List(?List(?a))`. A eliminação destas restrições é possível com a abordagem descrita na próxima seção.

7.2 Recursão e não-terminação de programas

O uso de recursão, tipos infinitos e a possibilidade de não-terminação de programas exigem o uso de um domínio semântico diferente, tanto nas especificações escritas em HasCasl quanto nas provas escritas para a ferramenta Isabelle. Este domínio representa um subconjunto de tipos onde a noção de especificação executável é permitida, ou seja, onde as especificações podem ser interpretadas como programas escritos em linguagens de programação funcional.

O domínio de especificações executáveis em HasCasl pode ser usado através das classes `Cpo` e `Cppo`, definidas na especificação `Recursion` da biblioteca `HasCASL/Metatheory/Recursion` e transcritos a seguir:

```
spec Recursion = Ord with Ord |-> InfOrd, __<=__ |-> __<<=__ and Nat then
class  Cpo < InfOrd {
var    a: Cpo
fun    __<<=__ : Pred (a * a)
op     undefined: ?a
. not def (undefined: ?a)
type   Chain a = {s: Nat ->? a . forall n: Nat . def s n => s n <<= s (n + 1)}
fun    sup: Chain a ->? a
var    x: ?a; c: Chain a
. sup c <<=[?a] x <=> forall n: Nat . c n <<=[?a] x
}
```

```
class Cppo < Cpo {
var    a : Cppo
fun    bottom : a
. bottom <<= x
}

class FlatCpo < Cpo {
vars   a : FlatCpo; x, y: a
. x <<= y => x = y
}

vars a, b: Cpo; c: Cppo; x, y: a; z, w: b
type instance __*__ : +Cpo -> +Cpo -> Cpo
type instance __*__ : +Cppo -> +Cppo -> Cppo

type instance Unit : Cppo
type instance Unit : FlatCpo

type a -->? b = { f : a ->? b . forall c: Chain a .
                 sup ((\ n: Nat . f (c n)) as Chain b) = f (sup c) }

type a --> b = { f : a -->? b . f in a -> b }

type instance __-->?__ : -Cpo -> +Cpo -> Cppo
var  f, g: a -->? b
. f <<= g <=> forall x: a . def (f x) => f x <<= g x

type instance __-->__ : -Cpo -> +Cpo -> Cpo
var  f, g: a --> b
. f <<= g <=> forall x: a . f x <<= g x
```

```
type instance __-->__ : -Cpo -> +Cppo -> Cppo
. bottom[a --> c] = \ x: a . bottom[c]

then %def

var c: Cppo
fun Y : (c --> c) --> c
var f : c --> c; x: c; P : Pred c
. f(Y f) = Y f
. f x = x => Y f <<= x
. P bottom /\ (forall x : c . P x => P (f x)) => P (Y f) %implied
```

Os tipos de dados utilizados nas especificações devem ser instâncias das classes `Cpo` ou `Cppo` e, quando polimórficos, devem estar definidos sobre variáveis de tipos que sejam instâncias destas classes. Um tipo nesta semântica pode ser definido com o uso da palavra-chave `free domain`, como a seguir:

```
var a : Cpo
free domain List a ::= Nil | Cons a (List a)
```

O tipo de dado definido acima representa uma lista finita de elementos cujo tipo é instância da classe `Cpo`. Esta lista pode ser utilizada em funções recursivas, embora ainda não capture as noções de listas com avaliação preguiçosa e nem de listas infinitas.

Para representar listas com avaliação preguiçosa e listas infinitas, deve-se definir o tipo de dado da seguinte forma:

```
var a : Cpo
free domain LList a ::= Nil | Cons a ?(LList a)
```

Então, listas finitas com avaliação preguiçosa podem ser obtidas mapeando-se um elemento do tipo `a` para uma lista indefinida (com avaliação preguiçosa). As listas infinitas podem ser criadas através de uma cadeia de listas finitas com avaliação preguiçosa.

Neste domínio, funções recursivas podem ser definidas como equações recursivas no estilo de linguagens de programação funcional, tal como Haskell, bastando defini-las após a palavra-chave `program`. A tradução destas equações para os termos que façam uso do operador de ponto fixo definido na especificação `Recursion` é implícita.

Neste domínio semântico pode-se definir funções com avaliação estrita ou preguiçosa, tal como no domínio padrão de HasCasl. A definição de funções com avaliação preguiçosa (como sugerido na seção anterior) dentro deste domínio semântico produz especificações com comportamento equivalente ao comportamento encontrado na linguagem Haskell.

Para escrever provas na ferramenta Isabelle dentro deste domínio é necessário alterar a lógica utilizada de HOL para HOLCF. O suporte à tradução para a teoria HOLCF pela ferramenta Hets ainda está em desenvolvimento e a complexidade do uso desta lógica é muito maior do que a do uso da lógica HOL.

O uso desta abordagem para avaliação preguiçosa, pelo acima descrito, exigiria mais tempo do que o disponível para a confecção deste trabalho. Notadamente, a necessidade de aprender mais uma lógica associada ao provador Isabelle demandaria um tempo consideravemente grande. Dessa forma, optou-se por não avançar na aplicação desta abordagem.

Capítulo 8

Conclusões e Trabalhos Futuros

Referências Bibliográficas

[1] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS Vol. 2960 (IFIP Series). Springer, 2004.

[2] Emden R. Gansner and John H. Reppy. *The Standard ML Basis Library*. Cambridge University Press, New York, NY, USA, 2002. ISBN 0-52179-478-1. URL `http://www.standardml.org/Basis/`.

[3] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of extended ml: A gentle introduction. *Theoretical Computer Science*, 173(2): 445 – 484, 1997. ISSN 0304-3975. doi: DOI:10.1016/S0304-3975(96)00163-6. URL `http://www.sciencedirect.com/science/article/B6V1G-3SNTKND-1V/2/716415b6c5c8885d02928dbfe56624c7`.

[4] Donald Sannella and Andrzei Tarlecki. Algebraic methods for specification and formal development of programs. *ACM Comput. Surv.*, 31(3):10, September 1999. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/333580.333589.

[5] Lutz Schröder and Till Mossakowski. Hascasl: Integrated higher-order specification and program development. *Theoretical Computer Science*, 410(12-13):1217–1260, 2009.

Apêndice A

Listagem das Especificações com Avaliação Estrita Desenvolvidas em HasCASL

Este apêndice contém o código das especificações com avaliação estrita desenvolvidas neste trabalho com o uso da linguagem HasCasl. O arquivo-fonte pode ser reconstruído compiando-se todas as especificações aqui descritas, na ordem apresentada, em um arquivo *Prelude.hs*.

A.1 Cabeçalhos da Biblioteca *Prelude*

```
library Prelude

version 0.1

%authors: Glauber M. Cabral <glauber.sp@gmail.com>

%date: 19 Fev 2008


logic HasCASL
```

```
from Basic/Numbers get Nat, Int, Rat
from HasCASL/Metatheory/Monad get Functor, Monad
from Basic/CharactersAndStrings get Char |-> IChar
```

A.2 Especificação *Bool*

```
spec Bool = %mono
free type Bool ::= True | False
fun Not__ : Bool -> Bool
fun __&&__ : Bool * Bool -> Bool
fun __||__ : Bool * Bool -> Bool
fun otherwiseH: Bool
vars x,y: Bool
. Not(False) = True %(NotFalse)%
. Not(True) = False                %(NotTrue)%
. False && x = False               %(AndFalse)%
. True && x = x                    %(AndTrue)%
. x && y = y && x                  %(AndSym)%
. x || y = Not(Not(x) && Not(y))   %(OrDef)%
. otherwiseH = True                %(OtherwiseDef)%
%%
. Not x = True <=> x = False       %(NotFalse1)% %implied
. Not x = False <=> x = True       %(NotTrue1)% %implied
. not (x = True) <=> Not x = True  %(notNot1)% %implied
. not (x = False) <=> Not x = False %(notNot2)% %implied
end
```

A.3 Especificação *Eq*

```
spec Eq = Bool then
class Eq {
var a: Eq
fun __==__ : a * a -> Bool
fun __/=__ : a * a -> Bool
vars x,y,z: a
. x = y => (x == y) = True                              %(EqualTDef)%
. x == y = y == x                                       %(EqualSymDef)%
. (x == x) = True                                       %(EqualReflex)%
. (x == y) = True /\ (y == z) = True => (x == z) = True   %(EqualTransT)%
. (x /= y) = Not (x == y)                               %(DiffDef)%
. (x /= y) = (y /= x)                                   %(DiffSymDef)% %implied
. (x /= y) = True <=> Not (x == y) = True               %(DiffTDef)% %implied
. (x /= y) = False <=> (x == y) = True                  %(DiffFDef)% %implied
. (x == y) = False => not (x = y)           %(TE1)% %implied
         %% == and Not need to be related!
. Not (x == y) = True <=> (x == y) = False   %(TE2)% %implied
. Not (x == y) = False <=> (x == y) = True   %(TE3)% %implied
. not ((x == y) = True) <=> (x == y) = False %(TE4)% %implied
}
type instance Bool: Eq
. (True == True) = True             %(IBE1)% %implied
. (False == False) = True           %(IBE2)% %implied
. (False == True) = False           %(IBE3)%
. (True == False) = False           %(IBE4)% %implied
. (True /= False) = True            %(IBE5)% %implied
. (False /= True) = True            %(IBE6)% %implied
. Not (True == False) = True        %(IBE7)% %implied
. Not (Not (True == False)) = False    %(IBE8)% %implied
```

```
type instance Unit: Eq
. (() == ()) = True  %(IUE1)% %implied
. (() /= ()) = False %(IUE2)% %implied
end
```

A.4 Especificação *Ord*

```
spec Ord = Eq and Bool then
free type Ordering ::= LT | EQ | GT
type instance Ordering: Eq
. (LT == LT) = True   %(IOE01)% %implied
. (EQ == EQ) = True   %(IOE02)% %implied
. (GT == GT) = True   %(IOE03)% %implied
. (LT == EQ) = False  %(IOE04)%
. (LT == GT) = False  %(IOE05)%
. (EQ == GT) = False  %(IOE06)%
. (LT /= EQ) = True   %(IOE07)% %implied
. (LT /= GT) = True   %(IOE08)% %implied
. (EQ /= GT) = True   %(IOE09)% %implied
class Ord < Eq
{
 var a: Ord
 fun compare: a -> a -> Ordering
 fun __<__ : a * a -> Bool
 fun __>__ : a * a -> Bool
 fun __<=__ : a * a -> Bool
 fun __>=__ : a * a -> Bool
 fun min: a -> a -> a
 fun max: a -> a -> a
 var    x, y, z, w: a
```

```
%% Definitions for relational operations.
%% Axioms for <
 . (x == y) = True => (x < y) = False                      %(LeIrreflexivity)%
 . (x < y) = True => y < x = False                         %(LeTAsymmetry)% %implied
 . (x < y) = True /\ (y < z) = True => (x < z) = True      %(LeTTransitive)%
 . (x < y) = True \/ (y < x) = True \/ (x == y) = True     %(LeTTotal)%
%% Axioms for >
 . (x > y) = (y < x)                                    %(GeDef)%
 . (x == y) = True => (x > y) = False                   %(GeIrreflexivity)% %implied
 . (x > y) = True => (y > x) = False                    %(GeTAsymmetry)% %implied
 . ((x > y)  && (y > z)) = True => (x > z) = True       %(GeTTransitive)% %implied
 . (((x > y) || (y > x)) || (x == y)) = True            %(GeTTotal)% %implied
%% Axioms for <=
 . (x <= y) = (x < y) || (x == y)                        %(LeqDef)%
 . (x <= x) = True                                       %(LeqReflexivity)% %implied
 . ((x <= y) && (y <= z)) = True => (x <= z) = True      %(LeqTTransitive)% %implied
 . (x <= y) && (y <= x) = (x == y)                       %(LeqTTotal)% %implied
%% Axioms for >=
 . (x >= y) = ((x > y) || (x == y))                 %(GeqDef)%
 . (x >= x) = True                                  %(GeqReflexivity)% %implied
 . ((x >= y) && (y >= z)) = True => (x >= z) = True %(GeqTTransitive)% %implied
 . (x >= y) && (y >= x) = (x == y)                  %(GeqTTotal)% %implied
%% Relates == and ordering
 . (x == y) = True <=> (x < y) = False /\ (x > y) = False  %(EqTSOrdRel)%
 . (x == y) = False <=> (x < y) = True \/ (x > y) = True   %(EqFSOrdRel)%
 . (x == y) = True <=> (x <= y) = True /\ (x >= y) = True  %(EqTOrdRel)%
 . (x == y) = False <=> (x <= y) = True \/ (x >= y) = True %(EqFOrdRel)%
 . (x == y) = True /\ (y < z) = True => (x < z) = True     %(EqTOrdTSubstE)%
 . (x == y) = True /\ (y < z) = False => (x < z) = False   %(EqTOrdFSubstE)%
 . (x == y) = True /\ (z < y) = True => (z < x) = True     %(EqTOrdTSubstD)%
 . (x == y) = True /\ (z < y) = False => (z < x) = False   %(EqTOrdFSubstD)%
```

```
 . (x < y) = True <=> (x > y) = False /\ (x == y) = False %(LeTGeFEqFRel)%
 . (x < y) = False <=> (x > y) = True \/ (x == y) = True  %(LeFGeTEqTRel)%
%% Relates all the ordering operators with True as result.
 . (x < y) = True <=> (y > x) = True     %(LeTGeTRel)% %implied
 . (x < y) = False <=> (y > x) = False   %(LeFGeFRel)% %implied
 . (x <= y) = True <=> (y >= x) = True   %(LeqTGetTRel)% %implied
 . (x <= y) = False <=> (y >= x) = False %(LeqFGetFRel)% %implied
 . (x > y) = True <=> (y < x) = True     %(GeTLeTRel)% %implied
 . (x > y) = False <=> (y < x) = False   %(GeFLeFRel)% %implied
 . (x >= y) = True <=> (y <= x) = True   %(GeqTLeqTRel)% %implied
 . (x >= y) = False <=> (y <= x) = False %(GeqFLeqFRel)% %implied
%%
 . (x <= y) = True <=> (x > y) = False                    %(LeqTGeFRel)% %implied
 . (x <= y) = False <=> (x > y) = True                    %(LeqFGeTRel)% %implied
 . (x > y) = True <=> (x < y) = False /\ (x == y) = False %(GeTLeFEqFRel)% %implied
 . (x > y) = False <=> (x < y) = True \/ (x == y) = True  %(GeFLeTEqTRel)% %implied
 . (x >= y) = True <=> (x < y) = False                    %(GeqTLeFRel)% %implied
 . (x >= y) = False <=> (x < y) = True                    %(GeqFLeTRel)% %implied
%%
 . (x <= y) = True <=> (x < y) = True \/ (x == y) = True    %(LeqTLeTEqTRel)% %implied
 . (x <= y) = False <=> (x < y) = False /\ (x == y) = False %(LeqFLeFEqFRel)% %implied
 . (x >= y) = True <=> (x > y) = True \/ (x == y) = True    %(GeqTGeTEqTRel)% %implied
 . (x >= y) = False <=> (x > y) = False /\ (x == y) = False %(GeqFGeFEqFRel)% %implied
%%
%% Implied True - False relations.
 . (x < y) = True <=> (x >= y) = False  %(LeTGeqFRel)% %implied
 . (x > y) = True <=> (x <= y) = False  %(GeTLeqFRel)% %implied
 . (x < y) = (x <= y) && (x /= y)       %(LeLeqDiff)% %implied
%% Definitions with compare
%% Definitions to compare, max and min using relational operations.
 . (compare x y == LT) = (x < y)                        %(CmpLTDef)%
```

```
 . (compare x y == EQ) = (x == y)                    %(CmpEQDef)%
 . (compare x y == GT) = (x > y)                     %(CmpGTDef)%
%% Define min, max
 . (max x y == y) = (x <= y)                        %(MaxYDef)%
 . (max x y == x) = (y <= x)                        %(MaxXDef)%
 . (min x y == x) = (x <= y)                        %(MinXDef)%
 . (min x y == y) = (y <= x)                        %(MinYDef)%
 . (max x y == y) = (max y x == y)                  %(MaxSym)% %implied
 . (min x y == y) = (min y x == y)                  %(MinSym)% %implied
}
%% Theorems
 . (x == y) = True \/ (x < y) = True <=> (x <= y) = True                  %(T01)% %implied
 . (x == y) = True  => (x < y) = False                                    %(T02)% %implied
 . Not (Not (x < y)) = True \/ Not (x < y) = True                          %(T03)% %implied
 . (x < y) = True => Not (x == y) = True                                  %(T04)% %implied
 . (x < y) = True /\ (y < z) = True /\ (z < w) = True => (x < w) = True   %(T05)% %implied
 . (z < x) = True => Not (x < z) = True                                   %(T06)% %implied
 . (x < y) = True <=> (y > x) = True                                      %(T07)% %implied
 type instance Ordering: Ord
 . (LT < EQ) = True                     %(IO013)%
 . (EQ < GT) = True                     %(IO014)%
 . (LT < GT) = True                     %(IO015)%
 . (LT <= EQ) = True                    %(IO016)% %implied
 . (EQ <= GT) = True                    %(IO017)% %implied
 . (LT <= GT) = True                    %(IO018)% %implied
 . (EQ >= LT) = True                    %(IO019)% %implied
 . (GT >= EQ) = True                    %(IO020)% %implied
 . (GT >= LT) = True                    %(IO021)% %implied
 . (EQ > LT) = True                     %(IO022)% %implied
 . (GT > EQ) = True                     %(IO023)% %implied
 . (GT > LT) = True                     %(IO024)% %implied
```

```
 . (max LT EQ == EQ) = True             %(IO025)% %implied
 . (max EQ GT == GT) = True             %(IO026)% %implied
 . (max LT GT == GT) = True             %(IO027)% %implied
 . (min LT EQ == LT) = True             %(IO028)% %implied
 . (min EQ GT == EQ) = True             %(IO029)% %implied
 . (min LT GT == LT) = True             %(IO030)% %implied
 . (compare LT LT == EQ) = True         %(IO031)% %implied
 . (compare EQ EQ == EQ) = True         %(IO032)% %implied
 . (compare GT GT == EQ) = True         %(IO033)% %implied
 type instance Bool: Ord
 . (False < True) = True                %(IBO5)%
 . (False >= True) = False              %(IBO6)% %implied
 . (True >= False) = True               %(IBO7)% %implied
 . (True < False) = False               %(IBO8)% %implied
 . (max False True == True) = True      %(IBO9)% %implied
 . (min False True == False) = True     %(IBO10)% %implied
 . (compare True True == EQ) = True     %(IBO11)% %implied
 . (compare False False == EQ) = True   %(IBO12)% %implied
 type instance Unit: Ord
 . (() <= ()) = True                    %(IUO01)% %implied
 . (() <  ()) = False                   %(IUO02)% %implied
 . (() >= ()) = True                    %(IUO03)% %implied
 . (() > ()) = False                    %(IUO04)% %implied
 . (max () () == ()) = True             %(IUO05)% %implied
 . (min () () == ()) = True             %(IUO06)% %implied
 . (compare () () == EQ) = True         %(IUO07)% %implied
end
```

A.5 Especificação *Maybe*

```
spec Maybe = Eq and Ord then

var a,b,c : Type;

    e : Eq;

    o : Ord;

free type Maybe a ::= Just a | Nothing

var x : a;

    y : b;

    ma : Maybe a;

    f : a -> b

fun maybe : b -> (a -> b) -> Maybe a -> b

. maybe y f (Just x: Maybe a) = f x                          %(MaybeJustDef)%

. maybe y f (Nothing: Maybe a) = y                           %(MaybeNothingDef)%

type instance Maybe e: Eq

var x,y : e;

. (Just x == Just y) = True <=> (x == y) = True        %(IME01)%

. ((Nothing : Maybe e) == (Nothing: Maybe e)) = True   %(IME02)% %implied

. Just x == Nothing = False                            %(IME03)%

type instance Maybe o: Ord

var x,y : o;

. (Nothing < Just x) = True                            %(IMO01)%

. (Just x < Just y) = (x < y)                          %(IMO02)%

. (Nothing >= Just x) = False                          %(IMO03)% %implied

. (Just x >= Nothing) = True                           %(IMO04)% %implied

. (Just x < Nothing) = False                           %(IMO05)% %implied

. (compare Nothing (Just x) == EQ)

     = (Nothing == (Just x))                           %(IMO06)% %implied

. (compare Nothing (Just x) == LT)

     = (Nothing < (Just x))                            %(IMO07)% %implied

. (compare Nothing (Just x) == GT)
```

```
    = (Nothing > (Just x))                          %(IMO08)% %implied
. (Nothing <= (Just x))
    = (max Nothing (Just x) == (Just x))            %(IMO09)% %implied
. ((Just x) <= Nothing)
    = (max Nothing (Just x) == Nothing)             %(IMO10)% %implied
. (Nothing <= (Just x))
    = (min Nothing (Just x) == Nothing)             %(IMO11)% %implied
. ((Just x) <= Nothing)
    = (min Nothing (Just x) == (Just x))            %(IMO12)% %implied
end
```

A.6 Especificação *MaybeMonad*

```
spec MaybeMonad = Maybe and Monad then
var a,b,c : Type;
         e : Eq;
         o : Ord;
type instance Maybe: Functor
vars  x: Maybe a;
      f: a -> b;
      g: b -> c
. map (\ y: a .! y) x = x                          %(IMF01)% %implied
. map (\ y: a .! g (f y)) x = map g (map f x)      %(IMF02)% %implied
type instance Maybe: Monad
vars  x, y: a;
      p: Maybe a;
      q: a ->? Maybe b;
      r: b ->? Maybe c;
      f: a ->? b
. def q x => ret x >>= q = q x                     %(IMM01)% %implied
```

```
. p >>= (\ x: a . ret (f x) >>= r)
     = p >>= \ x: a . r (f x)                            %(IMM02)% %implied
. p >>= ret = p                                          %(IMM03)% %implied
. (p >>= q) >>= r = p >>= \ x: a . q x >>= r             %(IMM04)% %implied
. (ret x : Maybe a) = ret y => x = y                     %(IMM05)% %implied
var x : Maybe a;
    f : a -> b;
. map f x = x >>= (\ y:a . ret (f y))                     %(T01)% %implied
end
```

A.7 Especificação *Either*

```
spec Either = Eq and Ord then
var a, b, c : Type; e, ee : Eq; o, oo : Ord;
free type Either a b ::= Left a | Right b
var x : a; y : b; z : c; eab : Either a b; f : a -> c; g : b -> c
fun either : (a -> c) -> (b -> c) -> Either a b -> c
. either f g (Left x: Either a b) = f x                      %(EitherLeftDef)%
. either f g (Right y: Either a b) = g y                     %(EitherRightDef)%
type instance Either e ee: Eq
var x,y : e; z,w : ee;
. ((Left x : Either e ee) ==
   (Left y : Either e ee)) = (x == y)                        %(IEE01)%
. ((Right z : Either e ee) ==
   (Right w : Either e ee)) = (z == w)                       %(IEE02)%
. ((Left x : Either e ee) ==
   (Right z : Either e ee)) = False                          %(IEE03)%
type instance Either o oo: Ord
var x,y : o; z,w : oo;
. ((Left x : Either o oo) < (Right z : Either o oo))
```

```
    = True                                              %(IE001)%
. ((Left x : Either o oo) < (Left y : Either o oo))
    = (x < y)                                           %(IE002)%
. ((Right z : Either o oo) < (Right w : Either o oo))
    = (z < w)                                           %(IE003)%
. ((Left x : Either o oo) >= (Right z : Either o oo))
    = False                                             %(IE004)% %implied
. ((Right z : Either o oo) >= (Left x : Either o oo))
    = True                                              %(IE005)% %implied
. ((Right z : Either o oo) < (Left x : Either o oo))
    = False                                             %(IE006)% %implied
. (compare (Left x : Either o oo) (Right z : Either o oo) == EQ)
    = ((Left x) == (Right z))                           %(IE007)% %implied
. (compare (Left x : Either o oo) (Right z : Either o oo) == LT)
    = ((Left x) < (Right z))                            %(IE008)% %implied
. (compare (Left x : Either o oo) (Right z : Either o oo) == GT)
    = ((Left x) > (Right z))                            %(IE009)% %implied
. ((Left x : Either o oo) <= (Right z : Either o oo))
    = (max (Left x) (Right z) == (Right z))             %(IE010)% %implied
. ((Right z : Either o oo) <= (Left x : Either o oo))
    = (max (Left x) (Right z) == (Left x))              %(IE011)% %implied
. ((Left x : Either o oo) <= (Right z : Either o oo))
    = (min (Left x) (Right z) == (Left x))              %(IE012)% %implied
. ((Right z : Either o oo) <= (Left x : Either o oo))
    = (min (Left x) (Right z) == (Right z))             %(IE013)% %implied
end
```

A.8 Especificação *EitherFunctor*

```
spec EitherFunctor = Either and Functor then
var a, b, c : Type;
    e, ee : Eq;
    o, oo : Ord;
type instance Either a: Functor
vars x: Either c a;
     f: a -> b;
     g: b -> c
. map (\ y: a .! y) x = x                         %(IEF01)% %implied
. map (\ y: a .! g (f y)) x = map g (map f x)      %(IEF02)% %implied
end
```

A.9 Especificação *Composition*

```
spec Composition =
vars a,b,c : Type
fun __o__ : (b -> c) * (a -> b) -> (a -> c);
vars a,b,c : Type; y:a;
     f : b -> c;
     g : a -> b
     . ((f o g) y) = f (g y)                 %(Comp1)%
end
```

A.10 Especificação *Function*

```
spec Function = Composition then
var a,b,c: Type;
    x: a;
    y: b;
```

```
    f: a -> b -> c;
    g: (a * b) -> c
fun id: a -> a
fun flip: (a -> b -> c) -> b -> a -> c
fun fst: (a * b) -> a
fun snd: (a * b) -> b
fun curry: ((a * b) -> c) -> a -> b -> c
fun uncurry: (a -> b -> c) -> (a * b) -> c
. id x = x                      %(IdDef)%
. flip f y x = f x y            %(FlipDef)%
. fst (x, y) = x                %(FstDef)%
. snd (x, y) = y                %(SndDef)%
. curry g x y = g (x, y)        %(CurryDef)%
. uncurry f (x,y) = f x y       %(UncurryDef)%
end
```

A.11 Especificação *ListNoNumbers*

```
spec ListNoNumbers = Function and Ord then
var a : Type
free type List a ::= Nil | Cons a (List a)
var a,b : Type
fun head : List a ->? a;
fun tail : List a ->? List a;
fun foldr : (a -> b -> b) -> b -> List a -> b;
fun foldl : (a -> b -> a) -> a -> List b -> a;
fun map : (a -> b) -> List a -> List b;
fun filter : (a -> Bool) -> List a -> List a;
fun __++__ : List a * List a -> List a;
fun zip : List a -> List b -> List (a * b);
```

```
fun unzip : List (a * b) -> (List a * List b)
vars a,b : Type;
     f : a -> b -> b;
     g : a -> b -> a;
     h : a -> b;
     p : a -> Bool;
     x,y,t : a;
     xs,ys,l : List a;
     z,s : b;
     zs : List b;
     ps : List (a * b)
. not def head (Nil : List a)                               %(NotDefHead)%
. head (Cons x xs) = x                                      %(HeadDef)%
. not def tail (Nil : List a)                               %(NotDefTail)%
. tail (Cons x xs) = xs                                     %(TailDef)%
. foldr f s Nil = s                                         %(FoldrNil)%
. foldr f s (Cons x xs)
     = f x (foldr f s xs)                                   %(FoldrCons)%
. foldl g t Nil = t                                         %(FoldlNil)%
. foldl g t (Cons z zs)
     = foldl g (g t z) zs                                   %(FoldlCons)%
. map h Nil = Nil                                           %(MapNil)%
. map h (Cons x xs)
     = (Cons (h x) (map h xs))                              %(MapCons)%
. Nil ++ l = l                                              %(++Nil)%
. (Cons x xs) ++ l = Cons x (xs ++ l)                       %(++Cons)%
. filter p Nil = Nil                                        %(FilterNil)%
. p x = True
     => filter p (Cons x xs) = Cons x (filter p xs)         %(FilterConsT)%
. p x = False
     => filter p (Cons x xs) = filter p xs                  %(FilterConsF)%
```

```
. zip (Nil : List a) l = Nil                                  %(ZipNil)%
. l = Nil
     => zip (Cons x xs) l = Nil                               %(ZipConsNil)%
. l = (Cons y ys)
     => zip (Cons x xs) l = Cons (x,y) (zip xs ys)            %(ZipConsCons)%
. unzip (Nil : List (a * b)) = (Nil, Nil)                     %(UnzipNil)%
. unzip (Cons (x,z) ps) = let (ys, zs) = unzip ps in
     (Cons x ys, Cons z zs)                                   %(UnzipCons)%
then
var a : Eq; x,y: a; xs, ys: List a
type instance List a: Eq
. ((Nil: List a) == (Nil: List a)) = True                     %(ILE01)% %implied
. ((Cons x xs) == (Cons y ys)) = ((x == y) && (xs == ys))     %(ILE02)%
var b : Ord; z,w: b; zs, ws: List b
type instance List b: Ord
. ((Nil: List b) < (Nil: List b)) = False                     %(ILO01)% %implied
. ((Nil: List b) <= (Nil: List b)) = True                     %(ILO02)% %implied
. ((Nil: List b) > (Nil: List b)) = False                     %(ILO03)% %implied
. ((Nil: List b) >= (Nil: List b)) = True                     %(ILO04)% %implied
. (z < w) = True => ((Cons z zs) < (Cons w ws)) = True        %(ILO05)%
. (z == w) = True => ((Cons z zs) < (Cons w ws)) = (zs < ws) %(ILO06)%
. (z < w) = False /\ (z == w) = False
     => ((Cons z zs) < (Cons w ws)) = False                   %(ILO07)%
. ((Cons z zs) <= (Cons w ws))
     = ((Cons z zs) < (Cons w ws))
          || ((Cons z zs) == (Cons w ws))                     %(ILO08)% %implied
. ((Cons z zs) > (Cons w ws))
     = ((Cons w ws) < (Cons z zs))                            %(ILO09)% %implied
. ((Cons z zs) >= (Cons w ws))
     = ((Cons z zs) > (Cons w ws))
          || ((Cons z zs) == (Cons w ws))                     %(ILO10)% %implied
```

```
. (compare (Nil: List b) (Nil: List b) == EQ)
     = ((Nil: List b) == (Nil: List b))                  %(ILO11)% %implied
. (compare (Nil: List b) (Nil: List b) == LT)
     = ((Nil: List b) < (Nil: List b))                   %(ILO12)% %implied
. (compare (Nil: List b) (Nil: List b) == GT)
     = ((Nil: List b) > (Nil: List b))                   %(ILO13)% %implied
. (compare (Cons z zs) (Cons w ws) == EQ)
     = ((Cons z zs) == (Cons w ws))                      %(ILO14)% %implied
. (compare (Cons z zs) (Cons w ws) == LT)
     = ((Cons z zs) < (Cons w ws))                       %(ILO15)% %implied
. (compare (Cons z zs) (Cons w ws) == GT)
     = ((Cons z zs) > (Cons w ws))                       %(ILO16)% %implied
. (max (Nil: List b) (Nil: List b) == (Nil: List b))
     = ((Nil: List b) <= (Nil: List b))                  %(ILO17)% %implied
. (min (Nil: List b) (Nil: List b) == (Nil: List b))
     = ((Nil: List b) <= (Nil: List b))                  %(ILO18)% %implied
. ((Cons z zs) <= (Cons w ws))
     = (max (Cons z zs) (Cons w ws) == (Cons w ws))      %(ILO19)% %implied
. ((Cons w ws) <= (Cons z zs))
     = (max (Cons z zs) (Cons w ws) == (Cons z zs))      %(ILO20)% %implied
. ((Cons z zs) <= (Cons w ws))
     = (min (Cons z zs) (Cons w ws) == (Cons z zs))      %(ILO21)% %implied
. ((Cons w ws) <= (Cons z zs))
     = (min (Cons z zs) (Cons w ws) == (Cons w ws))      %(ILO22)% %implied
then %implies
vars a,b,c : Ord;
     f : a -> b;
     g : b -> c;
     h : a -> a -> a;
     i : a -> b -> a;
     p : b -> Bool;
```

```
     x:a;
     y:b;
     xs,zs : List a;
     ys,ts : List b;
     z,e : a;
     xxs : List (List a)
. foldl i e (ys ++ ts)
     = foldl i (foldl i e ys) ts                          %(FoldlDecomp)%
. map f (xs ++ zs)
     = (map f xs) ++ (map f zs)                           %(MapDecomp)%
. map (g o f) xs = map g (map f xs)                       %(MapFunctor)%
. filter p (map f xs)
     = map f (filter (p o f) xs)                          %(FilterProm)%
then
vars a,b: Type;
     x,q: a;
     xs,qs: List a;
     y,z: b;
     ys,zs: List b;
     f: a -> a -> a;
     g: a -> b -> a;
     h: a -> b -> b;
fun init: List a ->? List a;
fun last: List a ->? a;
fun null: List a -> Bool;
fun reverse: List a -> List a;
fun foldr1: (a -> a -> a) -> List a ->? a;
fun foldl1: (a -> a -> a) -> List a ->? a;
fun scanl: (a -> b -> a) -> a -> List b -> List a
fun scanl1: (a -> a -> a) -> List a -> List a
fun scanr: (a -> b -> b) -> b -> List a -> List b
```

```
fun scanr1: (a -> a -> a) -> List a -> List a
. not def init (Nil: List a)                                    %(InitNil)%
. init (Cons x (Nil: List a)) = (Nil:List a)                    %(InitConsNil)%
. init (Cons x xs) = Cons x (init xs)                           %(InitConsCons)%
. not def last (Nil: List a)                                    %(LastNil)%
. last (Cons x (Nil: List a)) = x                               %(LastConsNil)%
. last (Cons x xs) = last xs                                    %(LastConsCons)%
. null (Nil:List a) = True                                      %(NullNil)%
. null (Cons x xs) = False                                      %(NullCons)%
. reverse (Nil: List a) = (Nil: List a)                         %(ReverseNil)%
. reverse (Cons x xs) = (reverse xs) ++ (Cons x (Nil: List a)) %(ReverseCons)%
. not def foldr1 f (Nil: List a)                                %(Foldr1Nil)%
. foldr1 f (Cons x (Nil: List a)) = x                           %(Foldr1ConsNil)%
. foldr1 f (Cons x xs) = f x (foldr1 f xs)                      %(Foldr1ConsCons)%
. not def foldl1 f (Nil: List a)                                %(Foldl1Nil)%
. foldl1 f (Cons x (Nil: List a)) = x                           %(Foldl1ConsNil)%
. foldl1 f (Cons x xs) = f x (foldr1 f xs)                      %(Foldl1ConsCons)%
. ys = Nil => scanl g q ys = Cons q Nil                            %(ScanlNil)%
. ys = (Cons z zs) => scanl g q ys = Cons q (scanl g (g q z) zs)  %(ScanlCons)%
. scanl1 f Nil = Nil                                               %(Scanl1Nil)%
. scanl1 f (Cons x xs) = scanl f x xs                              %(Scanl1Cons)%
. scanr h z Nil = Cons z Nil                                       %(ScanrNil)%
. (Cons y ys) = scanr h z xs
     => scanr h z (Cons x xs) = Cons (h x y)  (Cons y ys)          %(ScanrCons)%
. scanr1 f (Nil:List a) = (Nil:List a)                             %(Scanr1Nil)%
. scanr1 f (Cons x (Nil:List a)) = (Cons x (Nil:List a))           %(Scanr1ConsNil)%
. Cons q qs = scanr1 f xs
     => scanr1 f (Cons x xs) =  Cons (f x q) (Cons q qs)           %(Scanr1ConsCons)%
. last (scanl g x ys) = foldl g x ys                               %(ScanlProperty)% %implied
. head (scanr h y xs) = foldr h y xs                               %(ScanrProperty)% %implied
then
```

```
vars a,b,c : Type;
     b1,b2: Bool;
     d : Ord;
     x, y : a;
     xs, ys, zs : List a;
     xxs : List (List a);
     r, s : d;
     ds : List d;
     bs : List Bool;
     f : a -> a -> a;
     p, q : a -> Bool;
     g : a -> List b;
fun andL : List Bool -> Bool;
fun orL : List Bool -> Bool;
fun any : (a -> Bool) -> List a -> Bool;
fun all : (a -> Bool) -> List a -> Bool;
fun concatMap : (a -> List b) -> List a -> List b;
fun concat : List (List a) -> List a;
fun maximum : List d ->? d;
fun minimum : List d ->? d;
fun takeWhile : (a -> Bool) -> List a -> List a
fun dropWhile  : (a -> Bool) -> List a -> List a
fun span : (a -> Bool) -> List a -> (List a * List a)
fun break : (a -> Bool) -> List a -> (List a * List a)
. andL (Nil: List Bool) = True                          %(AndLNil)%
. andL (Cons b1 bs) = b1 && (andL bs)                   %(AndLCons)%
. orL (Nil: List Bool) = False                          %(OrLNil)%
. orL (Cons b1 bs) = b1 || (orL bs)                     %(OrLCons)%
. any p xs = orL (map p xs)                             %(AnyDef)%
. all p xs = andL (map p xs)                            %(AllDef)%
. concat xxs = foldr (curry __++__) (Nil: List a) xxs   %(ConcatDef)%
```

```
. concatMap g xs = concat (map g xs)                        %(ConcatMapDef)%
. not def maximum (Nil: List d)                             %(MaximunNil)%
. maximum ds = foldl1 max ds                                %(MaximumDef)%
. not def minimum (Nil: List d)                             %(MinimunNil)%
. minimum ds = foldl1 min ds                                %(MinimumDef)%
. takeWhile p (Nil: List a) = Nil: List a                   %(TakeWhileNil)%
. p x = True => takeWhile p (Cons x xs)
     = Cons x (takeWhile p xs)                              %(TakeWhileConsT)%
. p x = False => takeWhile p (Cons x xs) = Nil: List a      %(TakeWhileConsF)%
. dropWhile p (Nil: List a) = Nil: List a                   %(DropWhileNil)%
. p x = True => dropWhile p (Cons x xs) = dropWhile p xs    %(DropWhileConsT)%
. p x = False => dropWhile p (Cons x xs) = Cons x xs        %(DropWhileConsF)%
. span p (Nil: List a) = ((Nil: List a), (Nil: List a))     %(SpanNil)%
. p x = True => span p (Cons x xs)
     = let (ys, zs) = span p xs in
          ((Cons x ys), zs)                                 %(SpanConsT)%
. p x = False => span p (Cons x xs)
     = let (ys, zs) = span p xs in
          ((Nil: List a), (Cons x xs))                      %(SpanConsF)%
. span p xs = (takeWhile p xs, dropWhile p xs)              %(SpanThm)% %implied
. break p xs = let q = (Not__ o p) in span q xs             %(BreakDef)%
. break p xs = span (Not__ o p) xs                          %(BreakThm)% %implied
then
vars a,b,c : Type;
     d : Ord;
     e: Eq;
     x, y : a;
     xs, ys : List a;
     q, r : d;
     qs, rs : List d;
     s,t: e;
```

```
     ss,ts: List e;
     p: a -> Bool
fun insert: d -> List d -> List d
fun delete: e -> List e -> List e
fun select: (a -> Bool) -> a -> (List a * List a) -> (List a * List a)
fun partition: (a -> Bool) -> List a -> (List a * List a)
. insert q (Nil: List d) = Cons q Nil                       %(InsertNil)%
. (q <= r) = True => insert q (Cons r rs)
     = (Cons q (Cons r rs))                                 %(InsertCons1)%
. (q > r) = True => insert q (Cons r rs)
     = (Cons r (insert q rs))                               %(InsertCons2)%
. delete s (Nil: List e) = Nil                              %(DeleteNil)%
. (s == t) = True => delete s (Cons t ts) = ts              %(DeleteConsT)%
. (s == t) = False => delete s (Cons t ts)
     = (Cons t (delete s ts))                               %(DeleteConsF)%
. (p x) = True => select p x (xs, ys) = ((Cons x xs), ys)    %(SelectT)%
. (p x) = False => select p x (xs, ys) = (xs, (Cons x ys))   %(SelectF)%
. partition p xs = foldr (select p) ((Nil: List a),(Nil)) xs %(Partition)%
. partition p xs
    = (filter p xs, filter (Not__ o p) xs)          %(PartitionProp)% %implied
end
```

A.12 Especificação *NumericClasses*

```
spec NumericClasses = Ord and Nat and Int and Rat then
type instance Pos: Eq
type instance Pos: Ord
type instance Nat: Eq
type instance Nat: Ord
type instance Int: Eq
```

```
type instance Int: Ord
type instance Rat: Eq
type instance Rat: Ord

class Num < Eq {
 vars a: Num;
 x,y : a
 fun __+__: a * a -> a
 fun __*__: a * a -> a
 fun __-__: a * a -> a
 fun negate: a -> a
 fun abs: a -> a
 fun signum: a -> a
 fun fromInteger: Int -> a
}
vars a: Num;
     x,y : a
. (abs x) * (signum x) = x                  %(AbsSignumLaw)% %implied

type instance Pos: Num
vars a: Num;
     x,y: Pos;
     z: Int
. x + y = (__+__: Nat * Nat -> Nat) (x,y)    %(IPN01)%
. x * y = (__*__: Nat * Nat -> Nat) (x,y)    %(IPN02)%
. x - y = (__-!__: Nat * Nat -> Nat) (x,y)   %(IPN03)%
. negate x = 0 -! x                          %(IPN04)%
. (fun abs: a -> a) x = x                    %(IPN05)%
. signum x = 1                               %(IPN06)%
. fromInteger z = z as Pos                   %(IPN07)%
```

```
type instance Nat: Num
vars a: Num;
     x,y: Nat;
     z: Int
. x + y = (__+__: Nat * Nat -> Nat) (x,y)    %(INN01)%
. x * y = (__*__: Nat * Nat -> Nat) (x,y)    %(INN02)%
. x - y = (__-!__: Nat * Nat -> Nat) (x,y)   %(INN03)%
. negate x = 0 -! x                          %(INN04)%
. (fun abs: a -> a) x = x                    %(INN05)%
. signum x = 1                               %(INN06)%
. fromInteger z = z as Nat                   %(INN07)%

type instance Int: Num
vars a: Num;
     x,y: Int
. x + y = (__+__: Int * Int -> Int) (x,y)             %(IIN01)%
. x * y = (__*__: Int * Int -> Int) (x,y)             %(IIN02)%
. x - y = (__-__: Int * Int -> Int) (x,y)             %(IIN03)%
. negate x = 0 - x                                    %(IIN04)%
. (x >= 0) = True => (fun abs: a -> a) x = x          %(IIN05)%
. (x < 0) = True => (fun abs: a -> a) x = negate x    %(IIN06)%
. (x > 0) = True => signum x = 1                      %(IIN07)%
. (x == 0) = True => signum x = 0                     %(IIN07)%
. (x < 0) = True => signum x = - 1                    %(IIN08)%
. fromInteger x = x                                   %(IIN09)%

type instance Rat: Num
vars a: Num;
     x,y: Rat;
     z: Int
. x + y = (__+__: Rat * Rat -> Rat) (x,y)             %(IRN01)%
```

```
. x * y = (__*__: Rat * Rat -> Rat) (x,y)             %(IRN02)%
. x - y = (__-__: Rat * Rat -> Rat) (x,y)             %(IRN03)%
. negate x = 0 - x                                    %(IRN04)%
. (x >= 0) = True => (fun abs: a -> a) x = x          %(IRN05)%
. (x < 0) = True => (fun abs: a -> a) x = negate x    %(IRN06)%
. (x > 0) = True => signum x = 1                      %(IRN07)%
. (x == 0) = True => signum x = 0                     %(IRN07)%
. (x < 0) = True => signum x = - 1                    %(IRN08)%
. fromInteger z = z / 1                               %(IRN09)%

%% Integral should be subclass of Real and Enum that haven't been created yet
class Integral < Num
{
vars a: Integral;
fun __quot__, __rem__, __div__, __mod__: a * a -> a
fun quotRem, divMod: a -> a -> (a * a)
fun toInteger: a -> Int
}

type instance Nat: Integral
type instance Int: Integral
type instance Rat: Integral

%% Why can't I use x,y,z,w,r,s = a ?
vars a: Integral;
     x,y,z,w,r,s: a;
. (z,w) = quotRem x y => x quot y = z                                 %(IRI01)%
. (z,w) = quotRem x y => x rem y = w                                  %(IRI02)%
. (z,w) = divMod x y => x div y = z                                   %(IRI03)%
. (z,w) = divMod x y => x mod y = w                                   %(IRI04)%
. signum w = negate (signum y) /\ (z,w) = quotRem x y
```

```
    => divMod x y = (z - (fromInteger (toInteger (1:Nat))) , w + s)   %(IRI05)%
. not (signum w = negate (signum y)) /\ (z,w) = quotRem x y
    => divMod x y = (z, w)                                            %(IRI06)%

class Fractional < Num
{
vars a: Fractional
fun __/__ : a * a -> a
fun recip: a -> a
}

type instance Int: Fractional
type instance Rat: Fractional

vars a: Fractional;
     x,y: Int
. recip x = (1 / x)                                     %(IRI01)%
. x / y = x * (recip y)                                 %(IRI02)%

vars a: Fractional;
     x,y: Rat
. recip x = (1 / x)                                     %(IRF01)%
. x / y = x * (recip y)                                 %(IRF02)%

end
```

A.13 Especificação *ListWithNumbers*

```
spec ListWithNumbers = ListNoNumbers and NumericClasses then {
vars a,b: Type;
```

```
     c,d: Num;
     x,y : a;
     xs,ys : List a;
     n,nx : Int;
     z,w: Int;
     zs,ws: List Int
fun length: List a -> Int;
fun take: Int -> List a -> List a
fun drop: Int -> List a -> List a
fun splitAt: Int -> List a -> (List a * List a)
fun sum: List c -> c
fun sum': List c -> c -> c
fun product: List c -> c
fun product': List c -> c -> c
. length (Nil : List a) = 0                                  %(LengthNil)%
. length (Cons x xs) = (length xs) + 1                       %(LengthCons)%
. n <= 0 => take n xs = (Nil:List a)                         %(TakeNegative)%
. take n (Nil:List a) = (Nil:List a)                         %(TakeNil)%
. take n (Cons x xs) = Cons x (take (n-1) xs)                %(TakeCons)%
. n <= 0 => drop n xs = xs                                   %(DropNegative)%
. drop n (Nil:List a) = (Nil:List a)                         %(DropNil)%
. drop n (Cons x xs) =  drop (n-1) xs                        %(DropCons)%
. splitAt n xs =  (take n xs, drop n xs)                     %(SplitAt)%
. sum' (Nil: List Int) z = z                                 %(Sum'Nil)%
. sum' (Cons z zs) w
     = sum' zs ((fun __+__: c * c -> c)(w,z))                %(Sum'Cons)%
. sum zs = sum' zs 0                                         %(SumL)%
. product' (Nil: List Int) z = z                             %(Prod'Nil)%
. product' (Cons z zs) w
     = product' zs  ((fun __*__: c * c -> c)(w,z))           %(Prod'Cons)%
. product zs = product' zs 1                                 %(ProdL)%
```

```
then %implies
vars a,b,c : Ord;
     f : a -> b;
     g : b -> c;
     h : a -> a -> a;
     i : a -> b -> a;
     p : b -> Bool;
     x:a;
     y:b;
     xs,zs : List a;
     ys,ts : List b;
     z,e : a;
     xxs : List (List a)
. length (xs) = 0 <=> xs = Nil                                %(LengthNil1)%
. length (Nil : List a) = length ys
     => ys = (Nil : List b)                                   %(LengthEqualNil)%
. length (Cons x xs) = length (Cons y ys)
     => length xs = length ys                                 %(LengthEqualCons)%
. length xs = length ys
     => unzip (zip xs ys) = (xs, ys)                          %(ZipSpec)%
} hide sum', product'
end
```

A.14 Especificação *NumericFunctions*

```
spec NumericFunctions = Function and NumericClasses then {
var a,b: Type;
    a: Num;
    b: Integral;
    c: Fractional
```

```
fun subtract: a -> a -> a
fun even: b -> Bool
fun odd: b -> Bool
fun gcd: b -> b ->? b
fun lcm: b -> b -> b
fun gcd': b -> b -> b
fun __^__: a * b -> a
fun f: a -> b -> a
fun g: a -> b -> a -> a
fun __^^__: c * b -> c
vars a: Num;
     b: Integral;
     c: Fractional;
     x,y: Int;
     z,w: Int;
     r,s: Rat
. subtract x y = y - x                                         %(Subtract)%
. even z = (z rem (fromInteger 2)) == 0                        %(Even)%
. odd z = Not even z                                           %(Odd)%
. not def gcd 0 0                                              %(GgdUndef)%
. gcd z w = gcd' ((fun abs: a -> a) z)
     ((fun abs: a -> a) w)                                     %(Gcd)%
. gcd' z 0 = z                                                 %(Gcd'Zero)%
. gcd' z w = gcd' w (z rem w)                                  %(Gcd')%
. lcm z 0 = 0                                                  %(LcmVarZero)%
. lcm (toInteger 0) z = 0                                      %(LcmZeroVar)%
. lcm z w = (fun abs: a -> a)
     ((z quot ((fun gcd: b -> b ->? b) z w)) * w)              %(Lcm)%
. (z < 0) = True => not def(x ^ z)                             %(ExpUndef)%
. (z == 0) = True => x ^ z = 1                                 %(ExpOne)%
. (even y) = True => f x z = f (x * x) (z quot 2);             %(AuxF1)%
```

```
. (z == 1) = True => f x z = x;                         %(AuxF2)%
. (even y) = False /\ (z == 1) = False
     => f x z = g (x * x) ((y - 1) quot 2) x;           %(AuxF3)%
. (even y) = True => g x z w = g (x * x) (z quot 2) w;  %(AuxG1)%
. (y == 1) = True => g x z w = x * w;                   %(AuxG2)%
. (even y) = False /\ (y == 1) = False
     => g x z w = g (x * x) ((z - 1) quot 2) (x * w)    %(AuxG3)%
. (z < 0) = False /\ (z == 0) = False => x ^ z = f x z  %(Exp)%
} hide f,g
end
```

A.15 Especificação *Char*

```
spec Char = IChar and Ord and NumericClasses then
vars x, y: Char
type instance Char: Eq
. (ord(x) == ord(y)) = (x == y)                         %(ICE01)%
. Not(ord(x) == ord(y)) = (x /= y)                      %(ICE02)% %implied
type instance Char: Ord
%% Instance definition of <, <=, >, >=
. (ord(x) < ord(y)) = (x < y)                           %(ICO04)%
. (ord(x) <= ord(y)) = (x <= y)                         %(ICO05)% %implied
. (ord(x) > ord(y)) = (x > y)                           %(ICO06)% %implied
. (ord(x) >= ord(y)) = (x >= y)                         %(ICO07)% %implied
%% Instance definition of compare
. (compare x y == EQ) = (ord(x) == ord(y))              %(ICO01)% %implied
. (compare x y == LT) = (ord(x) < ord(y))               %(ICO02)% %implied
. (compare x y == GT) = (ord(x) > ord(y))               %(ICO03)% %implied
%% Instance defintion of min, max
. (ord(x) <= ord(y)) = (max x y == y)                   %(ICO08)% %implied
```

```
. (ord(y) <= ord(x)) = (max x y == x)                       %(IC009)% %implied
. (ord(x) <= ord(y)) = (min x y == x)                       %(IC010)% %implied
. (ord(y) <= ord(x)) = (min x y == y)                       %(IC011)% %implied
end
```

A.16 Especificação *String*

```
spec String = %mono
     ListNoNumbers and Char then
type String := List Char
vars a,b: String; x,y,z: Char; xs, ys: String
. x == y = True => ((Cons x xs) == (Cons y xs)) = True    %(StringT1)% %implied
. xs /= ys = True => ((Cons x ys) == (Cons y xs)) = False %(StringT2)% %implied
. (a /= b) = True =>  (a == b) = False                    %(StringT3)% %implied
. (x < y) = True =>  ((Cons x xs) < (Cons y xs)) = True   %(StringT4)% %implied
. (x < y) = True /\ (y < z) = True => ((Cons x (Cons z Nil))
         < (Cons x (Cons y Nil))) = False                 %(StringT5)% %implied
end
```

A.17 Especificação *MonadicList*

```
spec MonadicList = Monad and ListNoNumbers then
vars a,b: Type;
     m: Monad;
     f: a -> m b;
     ms: List (m a);
     k: m a -> m (List a) -> m (List a);
     n: m a;
     nn: m (List a);
     x: a;
```

```
     xs: List a;
fun sequence: List (m a) -> m (List a)
fun sequenceUnit: List (m a) -> m Unit
fun mapM: (a -> m b) -> List a -> m (List b)
fun mapMUnit: (a -> m b) -> List a -> m (List Unit)
. sequence ms = let
  k n nn = n >>= \ x:a. (nn >>= \ xs: List a . (ret (Cons x xs))) in
    foldr k (ret (Nil: List a)) ms                          %(SequenceListDef)%
end
```

A.18 Especificação *ExamplePrograms*

```
spec ExamplePrograms = ListNoNumbers then
var a: Ord;
    x,y: a;
    xs,ys: List a
fun quickSort: List a -> List a
fun insertionSort: List a -> List a
. quickSort (Nil: List a) = Nil                               %(QuickSortNil)%
. quickSort (Cons x xs)
     = ((quickSort (filter (\ y:a .! y < x) xs))
        ++ (Cons x Nil))
          ++ (quickSort (filter (\ y:a .! y >= x) xs))        %(QuickSortCons)%
. insertionSort (Nil: List a) = Nil                       %(InsertionSortNil)%
. insertionSort (Cons x xs) =
    insert x (insertionSort xs)                      %(InsertionSortConsCons)%
then %implies
var a: Ord;
    x,y: a;
    xs,ys: List a
```

```
. andL (Cons True (Cons True (Cons True Nil))) = True              %(Program01)%
. quickSort (Cons True (Cons False (Nil: List Bool)))
     = Cons False (Cons True Nil)                                  %(Program02)%
. insertionSort (Cons True (Cons False (Nil: List Bool)))
     = Cons False (Cons True Nil)                                  %(Program03)%
. insertionSort xs = quickSort xs                                  %(Program04)%
end
```

A.19 Especificação *SortingPrograms*

```
spec SortingPrograms = ListWithNumbers then
var a,b : Ord;
free type Split a b ::= Split b (List (List a))
var x,y,z,v,w: a;
    r,t: b;
    xs,ys,zs,vs,ws: List a;
    rs,ts: List b;
    xxs: List (List a);
    split: List a -> Split a b;
    join: Split a b -> List a;
    n: Nat
fun genSort: (List a -> Split a b) -> (Split a b -> List a) -> List a -> List a
fun splitInsertionSort: List b -> Split b b
fun joinInsertionSort: Split a a -> List a
fun insertionSort: List a -> List a
fun splitQuickSort: List a -> Split a a
fun joinQuickSort: Split b b -> List b
fun quickSort: List a -> List a
fun splitSelectionSort: List a -> Split a a
fun joinSelectionSort: Split b b -> List b
```

```
fun selectionSort: List a -> List a
fun splitMergeSort: List b -> Split b Unit
fun joinMergeSort: Split a Unit -> List a
fun merge: List a -> List a -> List a
fun mergeSort: List a -> List a
. xs = (Cons x (Cons y ys)) /\ split xs = Split r xxs
     => genSort split join xs
          = join (Split r (map (genSort split join) xxs))        %(GenSortT1)%
. xs = (Cons x (Cons y Nil)) /\ split xs = Split r xxs
     => genSort split join xs
          = join (Split r (map (genSort split join) xxs))        %(GenSortT2)%
. xs = (Cons x Nil) \/ xs = Nil
     => genSort split join xs = xs                                %(GenSortF)%
. splitInsertionSort (Cons x xs)
     = Split x (Cons xs (Nil: List (List a)))           %(SplitInsertionSort)%
. joinInsertionSort (Split x (Cons xs (Nil: List (List a))))
     = insert x xs                                        %(JoinInsertionSort)%
. insertionSort xs
     = genSort splitInsertionSort joinInsertionSort xs       %(InsertionSort)%
. splitQuickSort (Cons x xs)
     = let (ys, zs) = partition (\t:a .! x < t) xs
       in Split x (Cons ys (Cons zs Nil))                   %(SplitQuickSort)%
. joinQuickSort (Split x (Cons ys (Cons zs Nil)))
     = ys ++ (Cons x zs)                                     %(JoinQuickSort)%
. quickSort xs = genSort splitQuickSort joinQuickSort xs         %(QuickSort)%
. splitSelectionSort xs = let x = minimum xs
  in Split x (Cons (delete x xs) (Nil: List(List a)))    %(SplitSelectionSort)%
. joinSelectionSort (Split x (Cons xs Nil)) = (Cons x xs) %(JoinSelectionSort)%
. selectionSort xs
     = genSort splitSelectionSort joinSelectionSort xs       %(SelectionSort)%
. def((length xs) div 2) /\ n = ((length xs) div 2)
```

```
     => splitMergeSort xs = let (ys,zs) = splitAt n xs
        in Split () (Cons ys (Cons zs Nil))                %(SplitMergeSort)%
. xs = (Nil: List a) => merge xs ys = ys                         %(MergeNil)%
. xs = (Cons v vs) /\ ys = (Nil: List a)
     => merge xs ys = xs                                     %(MergeConsNil)%
. xs = (Cons v vs) /\ ys = (Cons w ws) /\ (v < w) = True
     => merge xs ys = Cons v (merge vs ys)                 %(MergeConsConsT)%
. xs = (Cons v vs) /\ ys = (Cons w ws) /\ (v < w) = False
     => merge xs ys = Cons w (merge xs ws)                 %(MergeConsConsF)%
. joinMergeSort (Split () (Cons ys (Cons zs Nil)))
     = merge ys zs                                          %(JoinMergeSort)%
. mergeSort xs = genSort splitMergeSort joinMergeSort xs        %(MergeSort)%
then
vars a: Ord;
     x,y: a;
     xs,ys: List a
preds __elem__ : a * List a;
      isOrdered: List a;
      permutation: List a * List a
. not x elem (Nil: List a)                                        %(ElemNil)%
. x elem (Cons y ys) <=> x = y \/ x elem ys                      %(ElemCons)%
. isOrdered (Nil: List a)                                    %(IsOrderedNil)%
. isOrdered (Cons x (Nil: List a))                          %(IsOrderedCons)%
. isOrdered (Cons x (Cons y ys))
     <=> (x <= y) = True /\ isOrdered(Cons y ys)        %(IsOrderedConsCons)%
. permutation ((Nil: List a), Nil)                         %(PermutationNil)%
. permutation (Cons x (Nil: List a), Cons y (Nil: List a))
     <=> x=y                                              %(PermutationCons)%
. permutation (Cons x xs, Cons y ys) <=>
     (x=y /\ permutation (xs, ys)) \/ (x elem ys
          /\ permutation(xs, Cons y (delete x ys)))   %(PermutationConsCons)%
```

```
then %implies
var a,b : Ord;
    xs, ys : List a;
. insertionSort xs = quickSort xs                    %(Theorem01)%
. insertionSort xs = mergeSort xs                    %(Theorem02)%
. insertionSort xs = selectionSort xs                %(Theorem03)%
. quickSort xs = mergeSort xs                        %(Theorem04)%
. quickSort xs = selectionSort xs                    %(Theorem05)%
. mergeSort xs = selectionSort xs                    %(Theorem06)%
. isOrdered(insertionSort xs)                        %(Theorem07)%
. isOrdered(quickSort xs)                            %(Theorem08)%
. isOrdered(mergeSort xs)                            %(Theorem09)%
. isOrdered(selectionSort xs)                        %(Theorem10)%
. permutation(xs, insertionSort xs)                  %(Theorem11)%
. permutation(xs, quickSort xs)                      %(Theorem12)%
. permutation(xs, mergeSort xs)                      %(Theorem13)%
. permutation(xs, selectionSort xs)                  %(Theorem14)%
end
```

Apêndice B

Listagem das Provas para Especificações com Avaliação Estrita Desenvolvidas em Isabelle/HOL

Este apêndice contém o código das provas para especificações com avaliação estrita desenvolvidas neste trabalho com o uso da linguagem HOL e verificadas com o provador de teorema Isabelle. Cada seção apresenta apenas os teoremas e o seus respectivos códigos de prova.

B.1 Prelude_Bool.thy

```
theorem NotFalse1 : "ALL x. Not' x = True' = (x = False')"
apply auto
apply(case_tac x)
apply auto
done
ML "Header.record \"NotFalse1\""
```

```
theorem NotTrue1 : "ALL x. Not' x = False' = (x = True')"
apply auto
apply(case_tac x)
apply auto
done
ML "Header.record \"NotTrue1\""
```

```
theorem notNot1 : "ALL x. (~ x = True') = (Not' x = True')"
apply(auto)
apply(case_tac x)
apply(auto)
done
ML "Header.record \"notNot1\""

theorem notNot2 : "ALL x. (~ x = False') = (Not' x = False')"
apply(auto)
apply(case_tac x)
apply(auto)
done
ML "Header.record \"notNot2\""

end
```

B.2 Prelude_Eq.thy

```
theorem DiffSymDef : "ALL x. ALL y. x /= y = y /= x"
apply(auto)
apply(simp add: DiffDef)
apply(simp add: EqualSymDef)
done
ML "Header.record \"DiffSymDef\""

theorem DiffTDef :
"ALL x. ALL y. x /= y = True' = (Not' (x ==' y) = True')"
apply(auto)
apply(simp add: DiffDef)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: DiffDef)
done
ML "Header.record \"DiffTDef\""

theorem DiffFDef :
"ALL x. ALL y. x /= y = False' = (x ==' y = True')"
apply(auto)
apply(simp add: DiffDef)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: DiffDef)
done
ML "Header.record \"DiffFDef\""

theorem TE1 : "ALL x. ALL y. x ==' y = False' --> ~ x = y"
by auto
ML "Header.record \"TE1\""

theorem TE2 :
"ALL x. ALL y. Not' (x ==' y) = True' = (x ==' y = False')"
apply auto
apply(case_tac "x ==' y")
apply auto
done
ML "Header.record \"TE2\""

theorem TE3 :
"ALL x. ALL y. Not' (x ==' y) = False' = (x ==' y = True')"
apply(auto)
apply(case_tac "x ==' y")
apply auto
done
ML "Header.record \"TE3\""

theorem TE4 :
"ALL x. ALL y. (~ x ==' y = True') = (x ==' y = False')"
apply auto
apply(case_tac "x ==' y")
apply auto
done

ML "Header.record \"TE4\""
```

```
theorem IBE1 : "True' ==' True' = True'"
by auto
ML "Header.record \"IBE1\""

theorem IBE2 : "False' ==' False' = True'"
by auto
ML "Header.record \"IBE2\""

theorem IBE4 : "True' ==' False' = False'"
apply(simp add: EqualSymDef)
done
ML "Header.record \"IBE4\""

theorem IBE5 : "True' /= False' = True'"
apply(simp add: DiffDef)
apply(simp add: IBE4)
done
ML "Header.record \"IBE5\""

theorem IBE6 : "False' /= True' = True'"
apply(simp add: DiffDef)
done
ML "Header.record \"IBE6\""

theorem IBE7 : "Not' (True' ==' False') = True'"
apply(simp add: IBE4)
done
ML "Header.record \"IBE7\""

theorem IBE8 : "Not' Not' (True' ==' False') = False'"
apply(simp add: IBE4)
done
ML "Header.record \"IBE8\""

theorem IUE1 : "() ==' () = True'"
by auto
ML "Header.record \"IUE1\""

theorem IUE2 : "() /= () = False'"
apply(simp add: DiffDef)
done
ML "Header.record \"IUE2\""

end
```

B.3 Prelude_Ord.thy

```
theorem IOE01 : "LT ==' LT = True'"
by auto
ML "Header.record \"IOE01\""

theorem IOE02 : "EQ ==' EQ = True'"
by auto
ML "Header.record \"IOE02\""

theorem IOE03 : "GT ==' GT = True'"
by auto
ML "Header.record \"IOE03\""

theorem IOE07 : "LT /= EQ = True'"
apply(simp add: DiffDef)
done
ML "Header.record \"IOE07\""

theorem IOE08 : "LT /= GT = True'"
apply(simp add: DiffDef)
done
ML "Header.record \"IOE08\""

theorem IOE09 : "EQ /= GT = True'"
apply(simp add: DiffDef)
done
ML "Header.record \"IOE09\""

lemma LeIrreflContra : " x <' x = True' ==> False"
```

```
by auto

theorem LeTAsymmetry :
"ALL x. ALL y. x <' y = True' --> y <' x = False'"
apply(auto)
apply(rule ccontr)
apply(simp add: notNot2 NotTrue1)
thm LeIrreflContra
apply(rule_tac x="x" in LeIrreflContra)
apply(rule_tac y = "y" in LeTTransitive)
by auto
ML "Header.record \"LeTAsymmetry\""

theorem GeIrreflexivity :
"ALL x. ALL y. x ==' y = True' --> x >' y = False'"
apply(auto)
apply(simp add: GeDef)
apply(simp add: EqualSymDef LeIrreflexivity)
done
ML "Header.record \"GeIrreflexivity\""

theorem GeTAsymmetry :
"ALL x. ALL y. x >' y = True' --> y >' x = False'"
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
done
ML "Header.record \"GeTAsymmetry\""

theorem GeTTransitive :
"ALL x.
 ALL y. ALL z. (x >' y) && (y >' z) = True'
--> x >' z = True'"
apply(auto)
apply(simp add: GeDef)
apply(rule_tac x="z" and y="y" and z="x" in  LeTTransitive)
apply(auto)
apply(case_tac  "z <' y")
apply(auto)
apply(case_tac  "y <' x")
apply(auto)
apply(case_tac  "y <' x")
apply(auto)
done
ML "Header.record \"GeTTransitive\""

theorem GeTTotal :
"ALL x. ALL y. ((x >' y) || (y >' x)) || (x ==' y) = True'"
apply(auto)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeFGeTEqTRel)
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
apply(simp add: EqualSymDef)
done
ML "Header.record \"GeTTotal\""

theorem LeqReflexivity : "ALL x. x <=' x = True'"
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
done
ML "Header.record \"LeqReflexivity\""

lemma EqualL1 [rule_format]:
"ALL x z.
 ((x ==' z) = True') & ((x ==' z) = False')
 \<longrightarrow> False"
by auto

lemma EqualL2 [rule_format]:
"ALL x. ALL y. ALL z.
((x ==' y) = True') & ((y ==' z) = True') \<longrightarrow>
((x ==' z) = False')\<longrightarrow> False"
apply(simp add: EqualL1)
apply(simp add: notNot2 NotTrue1)
```

```
apply(auto)
apply(rule EqualTransT)
apply(auto)
done

lemma Le1E [rule_format]:
"ALL x y z.
(y ==' x) = True' & (x <' z) = True'
 \<longrightarrow> (y <' z) = True'"
apply (auto)
apply(rule EqTOrdTSubstE)
apply(auto)
done

lemma Le2 [rule_format]:
"ALL x y.
(x <' y) = True' \<longrightarrow> (x <' y) = False'
\<longrightarrow> False"
by auto

lemma Le3E [rule_format]:
"ALL x y z.
(y ==' x) = True' & (x <' z) = True'
\<longrightarrow> (y <' z) = False'
\<longrightarrow> False"
apply (auto)
apply(rule Le2)
apply(rule EqTOrdTSubstE)
apply(auto)
done

lemma Le3D [rule_format]:
"ALL x y z.
(y ==' x) = True' & (z <' x) = True'
 \<longrightarrow> (z <' y) = False'
\<longrightarrow> False"
apply (auto)
apply(rule Le2)
apply(rule EqTOrdTSubstD)
apply(auto)
done

lemma Le4E [rule_format]:
"ALL x y z.
(y ==' x) = True' & (x <' z) = False'
 \<longrightarrow> (y <' z) = False'"
apply (auto)
apply(rule EqTOrdFSubstE)
apply(auto)
done

lemma Le4D [rule_format]:
"ALL x y z.
(y ==' x) = True' & (z <' x) = False'
 \<longrightarrow> (z <' y) = False'"
apply (auto)
apply(rule EqTOrdFSubstD)
apply(auto)
done

lemma Le5 [rule_format]:
"ALL x y.
(x <' y) = False' \<longrightarrow> (x <' y) = True'
\<longrightarrow> False"
by auto

lemma Le6E [rule_format]:
"ALL x y z.
(y ==' x) = True' & (x <' z) = False'
 \<longrightarrow> (y <' z) = True'
\<longrightarrow> False"
apply (auto)
apply(rule Le5)
apply(rule EqTOrdFSubstE)
apply(auto)
done

lemma Le7 [rule_format]:
"ALL x y.
x <' y = True' & x <' y = False' \<longrightarrow> False"
by auto

theorem LeqTTransitive :
"ALL x.
```

```
 ALL y. ALL z. (x <=' y) && (y <=' z) = True'
 --> x <=' z = True'"
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y <' z")
apply(auto)
apply(case_tac "y ==' z")
apply(auto)
apply(case_tac "x <' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(rule EqualL2)
apply(auto)
apply(simp add: NotFalse1 NotTrue1)
apply(case_tac "Not' (x <' z)")
apply(simp add: AndFalse)
apply(simp add: NotFalse1 NotTrue1)
apply(rule ccontr)
apply(simp add: notNot1 NotFalse1)
apply(erule Le2)
apply(rule Le4E)
apply(auto)
apply(simp add: EqualSymDef)
apply(case_tac "y <' z")
apply(auto)
apply(case_tac "y ==' z")
apply(auto)
apply(case_tac "x <' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(simp add: LeTGeFEqFRel)
apply(auto)
apply(simp add: LeFGeTEqTRel)
apply(simp add: EqTSOrdRel)
apply(simp add: EqFSOrdRel)
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTGeFEqFRel LeFGeTEqTRel)
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry LeIrreflexivity LeTTotal)
apply(simp add: GeDef)+
(*
apply(simp add: GeDef)
apply(simp add: GeDef)
*)
apply(simp add: EqualSymDef LeTGeFEqFRel LeFGeTEqTRel )
apply(simp add: GeDef)
(*The real proof seems to be in the next 3 lines.*)
apply(rule Le3E)
apply(auto)
apply(simp add: EqualSymDef)+
(*
apply(simp add: EqualSymDef)
apply(simp add: EqualSymDef)
apply(simp add: EqualSymDef)
*)
(*Verify until here.*)
(*The proof for the last goal.*)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x <' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(drule Le5)
apply(rule LeTTransitive)
apply(auto)
done
ML "Header.record \"LeqTTransitive\""

theorem LeqTTotal :
"ALL x. ALL y. (x <=' y) && (y <=' x) = x ==' y"
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
```

```
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: EqualSymDef)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: LeTAsymmetry)
done
ML "Header.record \"LeqTTotal\""

theorem GeqReflexivity : "ALL x. x >=' x = True'"
apply(auto)
apply(simp add: GeqDef)
apply(simp add: GeDef)
apply(simp add: OrDef)
done
ML "Header.record \"GeqReflexivity\""

theorem GeqTTransitive :
"ALL x.
 ALL y. ALL z. (x >=' y) && (y >=' z) = True'
 --> x >=' z = True'"
apply(auto)
apply(simp add: GeqDef)
apply(simp add: OrDef GeDef)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "z <' y")
apply(auto)
apply(case_tac "y ==' z")
apply(auto)
apply(case_tac "z <' x")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(rule EqualL2)
apply(auto)
apply(simp add: NotFalse1 NotTrue1)
apply(case_tac "Not' (z <' x)")
apply(simp add: AndFalse)
apply(simp add: NotFalse1 NotTrue1)
apply(rule ccontr)
apply(simp add: notNot1 NotFalse1)
apply(erule Le2)
apply(rule EqTOrdFSubstD)
apply(auto)
apply(simp add: EqualSymDef)
apply(case_tac "z <' y")
apply(auto)
apply(case_tac "y ==' z")
apply(auto)
apply(case_tac "z <' x")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(simp add: LeTGeFEqFRel)
apply(auto)
apply(simp add: LeFGeTEqTRel)
apply(simp add: EqTSOrdRel)
apply(simp add: EqFSOrdRel)
apply(auto)
apply(simp add: GeDef)+
apply(simp add: LeFGeTEqTRel LeTGeFEqFRel)
apply(auto)
apply(simp add: GeDef)
```

```
apply(simp add: LeTAsymmetry LeIrreflexivity LeTTotal)
apply(simp add: GeDef)+
apply(simp add: EqualSymDef LeTGeFEqFRel LeFGeTEqTRel )
apply(simp add: GeDef)
apply(rule Le3D)
apply(auto)
apply(simp add: EqualSymDef)+
apply(simp add: GeDef)+
apply(simp add: LeTAsymmetry)
apply(simp add: GeDef)+
apply(case_tac "z <' x")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(drule Le5)
apply(rule LeTTransitive)
apply(auto)
done
ML "Header.record \"GeqTTransitive\""

theorem GeqTTotal :
"ALL x. ALL y. (x >=' y) && (y >=' x) = x ==' y"
apply(auto)
apply(simp add: GeqDef)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: EqualSymDef)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
apply(simp add: EqualSymDef)
apply(case_tac "y >' x")
apply(auto)
done
ML "Header.record \"GeqTTotal\""

theorem LeTGeTRel :
"ALL x. ALL y. x <' y = True' = (y >' x = True')"
apply(auto)
apply(simp add: GeDef)
apply(simp add: GeDef)
done
ML "Header.record \"LeTGeTRel\""

theorem LeFGeFRel :
"ALL x. ALL y. x <' y = False' = (y >' x = False')"
apply(auto)
apply(simp add: GeDef)
apply(simp add: GeDef)
done
ML "Header.record \"LeFGeFRel\""

theorem LeqTGetTRel :
"ALL x. ALL y. x <=' y = True' = (y >=' x = True')"
apply(auto)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
```

```
apply(simp add: GeDef)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
done
ML "Header.record \"LeqTGetTRel\""

theorem LeqFGetFRel :
"ALL x. ALL y. x <=' y = False' = (y >=' x = False')"
apply(auto)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: GeDef)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: GeDef)
done
ML "Header.record \"LeqFGetFRel\""

theorem GeTLeTRel :
"ALL x. ALL y. x >' y = True' = (y <' x = True')"
apply(auto)
apply(simp add: GeDef)
apply(simp add: GeDef)
done
ML "Header.record \"GeTLeTRel\""

theorem GeFLeFRel :
"ALL x. ALL y. x >' y = False' = (y <' x = False')"
apply(auto)
apply(simp add: GeDef)
apply(simp add: GeDef)
done
ML "Header.record \"GeFLeFRel\""

theorem GeqTLeqTRel :
"ALL x. ALL y. x >=' y = True' = (y <=' x = True')"
apply(auto)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
```

```
apply(simp add: GeDef)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
done
ML "Header.record \"GeqTLeqTRel\""

theorem GeqFLeqFRel :
"ALL x. ALL y. x >=' y = False' = (y <=' x = False')"
apply(auto)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: GeDef)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: GeDef)
done
ML "Header.record \"GeqFLeqFRel\""

theorem LeqTGeFRel :
"ALL x. ALL y. x <=' y = True' = (x >' y = False')"
apply(auto)
apply(simp add: GeDef LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef LeIrreflexivity)
apply(simp add: LeTAsymmetry)
apply(simp add: LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqFSOrdRel)
done
ML "Header.record \"LeqTGeFRel\""

theorem LeqFGeTRel :
"ALL x. ALL y. x <=' y = False' = (x >' y = True')"
apply(auto)
apply(simp add: GeDef LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqFSOrdRel)
apply(simp add: GeDef)
apply(simp add: LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
```

```
apply(simp add: EqTSOrdRel)
apply(simp add: GeDef LeTAsymmetry)
done
ML "Header.record \"LeqFGeTRel\""

theorem GeTLeFEqFRel :
"ALL x.
 ALL y. x >' y = True' = (x <' y = False' & x ==' y = False')"
apply(auto)
apply(simp add: GeDef LeTAsymmetry)
apply(simp add: GeDef)
apply(simp add: EqFSOrdRel)
apply(auto)
apply(simp add: GeDef)
apply(simp add: EqFSOrdRel)
done
ML "Header.record \"GeTLeFEqFRel\""

theorem GeFLeTEqTRel :
"ALL x.
 ALL y. x >' y = False' = (x <' y = True' | x ==' y = True')"
apply(auto)
apply(simp add: LeTGeFEqFRel)
apply(simp add: notNot1)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
apply(simp add: GeDef)
apply(simp add: EqualSymDef LeIrreflexivity)
done
ML "Header.record \"GeFLeTEqTRel\""

theorem GeqTLeFRel :
"ALL x. ALL y. x >=' y = True' = (x <' y = False')"
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: EqFSOrdRel)
apply(simp add: GeDef)
done
ML "Header.record \"GeqTLeFRel\""

theorem GeqFLeTRel :
"ALL x. ALL y. x >=' y = False' = (x <' y = True')"
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: EqFSOrdRel)
apply(simp add: GeDef)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
done
ML "Header.record \"GeqFLeTRel\""

theorem LeqTLeTEqTRel :
"ALL x.
 ALL y. x <=' y = True'
= (x <' y = True' | x ==' y = True')"
apply(auto)
apply(simp add: LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
```

```
apply(auto)
apply(simp add: LeqDef OrDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"LeqTLeTEqTRel\""

theorem LeqFLeFEqFRel :
"ALL x.
 ALL y. x <=' y = False'
= (x <' y = False' & x ==' y = False')"
apply(auto)
apply(simp add: LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(simp add: LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"LeqFLeFEqFRel\""

theorem GeqTGeTEqTRel :
"ALL x.
 ALL y. x >=' y = True'
= (x >' y = True' | x ==' y = True')"
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeqDef OrDef)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
done
ML "Header.record \"GeqTGeTEqTRel\""

theorem GeqFGeFEqFRel :
"ALL x.
 ALL y. x >=' y = False'
= (x >' y = False' & x ==' y = False')"
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeqDef OrDef)
done
ML "Header.record \"GeqFGeFEqFRel\""

theorem LeTGeqFRel :
"ALL x. ALL y. x <' y = True' = (x >=' y = False')"
apply(auto)
apply(simp add: LeTGeFEqFRel)
apply(simp add: GeqDef)
apply(simp add: OrDef)
apply(simp add: GeqFGeFEqFRel)
apply(simp add: LeTGeFEqFRel)
done
ML "Header.record \"LeTGeqFRel\""

theorem GeTLeqFRel :
"ALL x. ALL y. x >' y = True' = (x <=' y = False')"
apply(auto)
apply(simp add: GeTLeFEqFRel)
apply(simp add: LeqDef)
apply(simp add: OrDef)
apply(simp add: LeqFLeFEqFRel)
apply(simp add: GeTLeFEqFRel)
done
ML "Header.record \"GeTLeqFRel\""

theorem LeLeqDiff : "ALL x. ALL y. x <' y
= (x <=' y) && (x /= y)"
apply(auto)
apply(simp add: LeqDef OrDef)
apply(case_tac "x <' y")
```

```
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "x /= y")
apply(auto)
apply(simp add: DiffDef)
apply(simp add: LeTGeFEqFRel)
apply(simp add: DiffDef)
done
ML "Header.record \"LeLeqDiff\""

theorem MaxSym : "ALL x. ALL y. X_max x y ==' y
= X_max y x ==' y"
by auto
ML "Header.record \"MaxSym\""

theorem MinSym : "ALL x. ALL y. X_min x y ==' y
= X_min y x ==' y"
by auto
ML "Header.record \"MinSym\""

theorem TO1 :
"ALL x.
 ALL y. (x ==' y = True' | x <' y = True')
= (x <=' y = True')"
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
done
ML "Header.record \"TO1\""

theorem TO2 : "ALL x. ALL y. x ==' y = True'
--> x <' y = False'"
by auto
ML "Header.record \"TO2\""

theorem TO3 :
"ALL x. ALL y. Not' Not' (x <' y)
= True' | Not' (x <' y) = True'"
apply(auto)
apply(case_tac "x <' y")
apply(auto)
done
ML "Header.record \"TO3\""

theorem TO4 :
"ALL x. ALL y. x <' y = True' --> Not' (x ==' y) = True'"
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
done
ML "Header.record \"TO4\""

theorem TO5 :
"ALL w.
 ALL x.
 ALL y.
 ALL z.
 (x <' y = True' & y <' z = True') & z <' w = True' -->
 x <' w = True'"
apply auto
apply(rule_tac y="y" in LeTTransitive)
apply auto
apply(rule_tac y="z" in LeTTransitive)
by auto
ML "Header.record \"TO5\""

theorem TO6 :
"ALL x. ALL z. z <' x = True' --> Not' (x <' z) = True'"
apply auto
apply(case_tac "x <' z")
apply auto
apply (simp add: LeTAsymmetry)
done
```

```
ML "Header.record \"T06\""

theorem T07 : "ALL x. ALL y. x <' y = True' = (y >' x = True')"
apply auto
apply(simp add: GeDef)+
done
ML "Header.record \"T07\""

theorem IO016 : "LT <=' EQ = True'"
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO016\""

theorem IO017 : "EQ <=' GT = True'"
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO017\""

theorem IO018 : "LT <=' GT = True'"
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO018\""

theorem IO019 : "EQ >=' LT = True'"
apply(simp add: GeqDef OrDef GeDef)
done
ML "Header.record \"IO019\""

theorem IO020 : "GT >=' EQ = True'"
apply(simp add: GeqDef OrDef GeDef)
done
ML "Header.record \"IO020\""

theorem IO021 : "GT >=' LT = True'"
apply(simp add: GeqDef OrDef GeDef)
done
ML "Header.record \"IO021\""

theorem IO022 : "EQ >' LT = True'"
apply(simp add: GeDef OrDef)
done
ML "Header.record \"IO022\""

theorem IO023 : "GT >' EQ = True'"
apply(simp add: GeDef OrDef)
done
ML "Header.record \"IO023\""

theorem IO024 : "GT >' LT = True'"
apply(simp add: GeDef OrDef)
done
ML "Header.record \"IO024\""

theorem IO025 : "X_max LT EQ ==' EQ = True'"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO025\""

theorem IO026 : "X_max EQ GT ==' GT = True'"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO026\""

theorem IO027 : "X_max LT GT ==' GT = True'"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO027\""

theorem IO028 : "X_min LT EQ ==' LT = True'"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO028\""

theorem IO029 : "X_min EQ GT ==' EQ = True'"
apply(simp add: MinXDef)
apply(simp add: LeqDef OrDef)
done

ML "Header.record \"IO029\""
```

```
theorem IO030 : "X_min LT GT ==' LT = True'"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO030\""

theorem IO031 : "compare LT LT ==' EQ = True'"
by auto
ML "Header.record \"IO031\""

theorem IO032 : "compare EQ EQ ==' EQ = True'"
by auto
ML "Header.record \"IO032\""

theorem IO033 : "compare GT GT ==' EQ = True'"
by auto
ML "Header.record \"IO033\""

theorem IB06 : "False' >=' True' = False'"
apply(simp add: GeqDef OrDef GeDef)
apply (case_tac "True' <' False'")
apply(auto)
apply(simp add: LeTGeFEqFRel)
apply(simp add: GeDef)
done
ML "Header.record \"IB06\""

theorem IB07 : "True' >=' False' = True'"
apply(simp add: GeqDef OrDef GeDef)
done
ML "Header.record \"IB07\""

theorem IB08 : "True' <' False' = False'"
apply(simp add: LeFGeTEqTRel)
apply(simp add: GeDef)
done
ML "Header.record \"IB08\""

theorem IB09 : "X_max False' True' ==' True' = True'"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IB09\""

theorem IB010 : "X_min False' True' ==' False' = True'"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IB010\""

theorem IB011 : "compare True' True' ==' EQ = True'"
by auto
ML "Header.record \"IB011\""

theorem IB012 : "compare False' False' ==' EQ = True'"
by auto
ML "Header.record \"IB012\""

theorem IUO01 : "() <=' () = True'"
apply (simp add: LeqDef OrDef)
done
ML "Header.record \"IUO01\""

theorem IUO02 : "() <' () = False'"
by auto
ML "Header.record \"IUO02\""

theorem IUO03 : "() >=' () = True'"
apply(simp add: GeqDef GeDef OrDef)
done
ML "Header.record \"IUO03\""

theorem IUO04 : "() >' () = False'"
apply(simp add: GeDef)
done
ML "Header.record \"IUO04\""

theorem IUO05 : "X_max () () ==' () = True'"
by auto
ML "Header.record \"IUO05\""

theorem IUO06 : "X_min () () ==' () = True'"
by auto
ML "Header.record \"IUO06\""
```

```
theorem IU007 : "compare () () ==' EQ = True'"
by auto
ML "Header.record \"IU007\""

end
```

B.4 Prelude_Maybe.thy

```
theorem IME02 : "Nothing ==' Nothing = True'"
by auto
ML "Header.record \"IME02\""

theorem IMO03 : "ALL x. Nothing >=' Just(x) = False'"
apply(rule allI)
apply(simp only: GeqDef)
apply(simp only: GeDef OrDef)
apply(case_tac "Just(x) <' Nothing")
apply(auto)
done
ML "Header.record \"IMO03\""

theorem IMO04 : "ALL x. Just(x) >=' Nothing = True'"
apply(rule allI)
apply(simp only: GeqDef)
apply(simp only: GeDef OrDef)
apply(case_tac "Nothing <' Just(x)")
apply(auto)
done
ML "Header.record \"IMO04\""

theorem IMO05 : "ALL x. Just(x) <' Nothing = False'"
apply(rule allI)
apply(case_tac "Just(x) <' Nothing")
apply(auto)
done
ML "Header.record \"IMO05\""

theorem IMO06 :
"ALL x. compare Nothing (Just(x)) ==' EQ
= Nothing ==' Just(x)"
by auto
ML "Header.record \"IMO06\""

theorem IMO07 :
"ALL x. compare Nothing (Just(x)) ==' LT
= Nothing <' Just(x)"
by auto
ML "Header.record \"IMO07\""

theorem IMO08 :
"ALL x. compare Nothing (Just(x)) ==' GT
= Nothing >' Just(x)"
apply(rule allI)+
apply(simp add: GeDef)
done
ML "Header.record \"IMO08\""

theorem IMO09 :
"ALL x. Nothing <=' Just(x)
= X_max Nothing (Just(x)) ==' Just(x)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IMO09\""

theorem IMO10 :
"ALL x. Just(x) <=' Nothing
= X_max Nothing (Just(x)) ==' Nothing"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IMO10\""

theorem IMO11 :
```

```
"ALL x. Nothing <=' Just(x)
= X_min Nothing (Just(x)) ==' Nothing"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IM011\""

theorem IM012 :
"ALL x. Just(x) <=' Nothing
= X_min Nothing (Just(x)) ==' Just(x)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IM012\""

end
```

B.5 Prelude_Either.thy

```
theorem IE004 : "ALL x. ALL z. Left'(x) >=' Right'(z)
 = False'"
apply(rule allI)
apply(simp only: GeqDef)
apply(simp only: GeDef OrDef)
apply(case_tac "Right'(y) <' Left'(x)")
apply(auto)
done
ML "Header.record \"IE004\""

theorem IE005 : "ALL x. ALL z. Right'(z) >=' Left'(x)
 = True'"
apply(rule allI)
apply(simp only: GeqDef)
apply(simp only: GeDef OrDef)
apply(case_tac "Left'(x) <' Right'(y)")
apply(auto)
done
ML "Header.record \"IE005\""

theorem IE006 : "ALL x. ALL z. Right'(z) <' Left'(x)
 = False'"
apply(rule allI)
apply(case_tac "Right'(y) <' Left'(x)")
apply(auto)
done
ML "Header.record \"IE006\""

theorem IE007 :
"ALL x.
 ALL z.
 compare (Left'(x)) (Right'(z)) ==' EQ
= Left'(x) ==' Right'(z)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE007\""

theorem IE008 :
"ALL x.
 ALL z.
 compare (Left'(x)) (Right'(z)) ==' LT
 = Left'(x) <' Right'(z)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE008\""

theorem IE009 :
"ALL x.
 ALL z.
 compare (Left'(x)) (Right'(z)) ==' GT
= Left'(x) >' Right'(z)"
apply(rule allI)+
apply(simp add: GeDef)
done
```

```
ML "Header.record \"IE009\""

theorem IE010 :
"ALL x.
 ALL z.
 Left'(x) <=' Right'(z) =
 X_max (Left'(x)) (Right'(z)) ==' Right'(z)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE010\""

theorem IE011 :
"ALL x.
 ALL z.
 Right'(z) <=' Left'(x)
= X_max (Left'(x)) (Right'(z)) ==' Left'(x)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE011\""

theorem IE012 :
"ALL x.
 ALL z.
 Left'(x) <=' Right'(z)
= X_min (Left'(x)) (Right'(z)) ==' Left'(x)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE012\""

theorem IE013 :
"ALL x.
 ALL z.
 Right'(z) <=' Left'(x) =
 X_min (Left'(x)) (Right'(z)) ==' Right'(z)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE013\""

end
```

B.6 Prelude_ListNoNumbers.thy

```
theorem PartitionProp :
"ALL p.
 ALL xs.
 partition p xs =
 (X_filter p xs, X_filter (X__o__X (Not__X, p)) xs)"
apply(auto)
apply(simp only: Partition)
apply(induct_tac xs)
apply(case_tac "p a")
apply(simp only: FoldrCons)
apply(simp only: FilterConsF)
apply(auto)
apply(simp add: FilterConsT)
apply(simp add: FoldrCons)
apply(simp only: FilterConsT)
done
ML "Header.record \"PartitionProp\""

end
```

B.7 Prelude_ListNoNumbers_ E1.thy

```
theorem SpanThm :
"ALL p. ALL xs. span p xs
 = (X_takeWhile p xs, X_dropWhile p xs)"
apply(auto)
apply(case_tac xs)
apply(auto)
apply(induct_tac List)
apply(case_tac "p a")
apply(simp add: TakeWhileConsF DropWhileConsF SpanConsF)
apply(case_tac "p aa")
apply(simp add: TakeWhileConsT DropWhileConsT
 SpanConsT TakeWhileConsF DropWhileConsF
 SpanConsF TakeWhileNil DropWhileNil SpanNil)+
apply(simp only: Let_def)
apply(simp add: split_def)
apply(case_tac "p a")
apply(simp add: TakeWhileConsT DropWhileConsT
 SpanConsT TakeWhileConsF
DropWhileConsF SpanConsF TakeWhileNil
 DropWhileNil SpanNil)+
done
ML "Header.record \"SpanThm\""

theorem BreakThm :
"ALL p. ALL xs. break p xs
= span (X__o__X (Not__X, p)) xs"
apply(auto)
apply(case_tac xs)
apply(auto)
apply(simp add: BreakDef)
apply(simp add: Let_def)
apply(simp add: BreakDef)
done
ML "Header.record \"BreakThm\""

end
```

B.8 Prelude_ListNoNumbers_ E4.thy

```
theorem ILE01 : "Nil' ==' Nil' = True'"
by auto
ML "Header.record \"ILE01\""

theorem ILO01 : "Nil' <'' Nil' = False'"
by auto
ML "Header.record \"ILO01\""

theorem ILO02 : "Nil' <='' Nil' = True'"
by auto
ML "Header.record \"ILO02\""

theorem ILO03 : "Nil' >'' Nil' = False'"
by auto
ML "Header.record \"ILO03\""

theorem ILO04 : "Nil' >='' Nil' = True'"
by auto
ML "Header.record \"ILO04\""

theorem ILO08 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons z zs <='' X_Cons w ws =
 (X_Cons z zs <'' X_Cons w ws)
|| (X_Cons z zs ==' X_Cons w ws)"
apply(rule allI)+
apply(simp only: LeqDef)
done
ML "Header.record \"ILO08\""

theorem ILO09 :
"ALL w.
 ALL ws.
```

```
 ALL z.
 ALL zs. X_Cons z zs >'' X_Cons w ws
= X_Cons w ws <'' X_Cons z zs"
apply(rule allI)+
apply(case_tac "X_Cons z zs >'' X_Cons w ws")
apply(simp only: GeFLeFRel)
apply(simp only: GeTLeTRel)
done
ML "Header.record \"IL009\""

theorem IL010 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons z zs >='' X_Cons w ws =
 (X_Cons z zs >'' X_Cons w ws)
|| (X_Cons z zs ==' X_Cons w ws)"
apply(rule allI)+
apply(simp only: GeqDef)
done
ML "Header.record \"IL010\""

theorem IL011 : "compare Nil' Nil' ==' EQ
 = Nil' ==' Nil'"
by auto
ML "Header.record \"IL011\""

theorem IL012 : "compare Nil' Nil' ==' LT
 = Nil' <'' Nil'"
by auto
ML "Header.record \"IL012\""

theorem IL013 : "compare Nil' Nil' ==' GT
 = Nil' >'' Nil'"
by auto
ML "Header.record \"IL013\""

theorem IL014 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 compare (X_Cons z zs) (X_Cons w ws) ==' EQ =
 X_Cons z zs ==' X_Cons w ws"
apply(rule allI)+
apply(simp only: CmpEQDef)
done
ML "Header.record \"IL014\""

theorem IL015 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 compare (X_Cons z zs) (X_Cons w ws) ==' LT =
 X_Cons z zs <'' X_Cons w ws"
apply(rule allI)+
apply(simp only: CmpLTDef)
done

ML "Header.record \"IL015\""

theorem IL016 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 compare (X_Cons z zs) (X_Cons w ws) ==' GT =
 X_Cons z zs >'' X_Cons w ws"
apply(rule allI)+
apply(simp only: CmpGTDef)
done
ML "Header.record \"IL016\""

theorem IL017 : "X_maxX2 Nil' Nil' ==' Nil'
 = Nil' <='' Nil'"
by auto
ML "Header.record \"IL017\""

theorem IL018 : "X_minX2 Nil' Nil' ==' Nil'
 = Nil' <='' Nil'"
by auto
ML "Header.record \"IL018\""
```

```
theorem IL019 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons z zs <='' X_Cons w ws =
 X_maxX2 (X_Cons z zs) (X_Cons w ws) ==' X_Cons w ws"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IL019\""

theorem IL020 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons w ws <='' X_Cons z zs =
 X_maxX2 (X_Cons z zs) (X_Cons w ws) ==' X_Cons z zs"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IL020\""

theorem IL021 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons z zs <='' X_Cons w ws =
 X_minX2 (X_Cons z zs) (X_Cons w ws) ==' X_Cons z zs"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IL021\""

theorem IL022 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons w ws <='' X_Cons z zs =
 X_minX2 (X_Cons z zs) (X_Cons w ws) ==' X_Cons w ws"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IL022\""

theorem FoldlDecomp :
"ALL e.
 ALL i.
 ALL ts.
 ALL ys. X_foldl i e (ys ++' ts)
= X_foldl i (X_foldl i e ys) ts"
oops
ML "Header.record \"FoldlDecomp\""

theorem MapDecomp :
"ALL f.
 ALL xs. ALL zs. X_map f (xs ++' zs)
= X_map f xs ++' X_map f zs"
apply(auto)
apply(induct_tac xs)
apply(auto)
apply(simp add: MapCons XPlusXPlusCons)
done
ML "Header.record \"MapDecomp\""

theorem MapFunctor :
"ALL f.
 ALL g. ALL xs. X_map (X__o__X (g, f)) xs
 = X_map g (X_map f xs)"
apply(auto)
apply(induct_tac xs)
apply(auto)
apply(simp add: MapNil MapCons Comp1)
done
ML "Header.record \"MapFunctor\""

theorem FilterProm :
"ALL f.
 ALL p.
 ALL xs.
```

```
 X_filter p (X_map f xs)
= X_map f (X_filter (X__o__X (p, f)) xs)"
apply(auto)
apply(induct_tac xs)
apply(auto)
apply(case_tac "p(f a)")
apply(auto)
apply(simp add: MapCons)
apply(simp add: FilterConsT)
apply(simp add: MapCons)
apply(simp add: FilterConsT)
done
ML "Header.record \"FilterProm\""

theorem LengthNil1 : "ALL xs. length'(xs)
= 0' = (xs = Nil')"
apply(auto)
apply(case_tac xs)
apply(auto)
done
ML "Header.record \"LengthNil1\""

theorem LengthEqualNil :
"ALL ys. length'(Nil') = length'(ys) --> ys = Nil'"
apply(auto)
apply(case_tac ys)
apply(auto)
done
ML "Header.record \"LengthEqualNil\""

theorem LengthEqualCons :
"ALL x.
 ALL xs.
 ALL y.
 ALL ys.
 length'(X_Cons x xs) = length'(X_Cons y ys) -->
 length'(xs) = length'(ys)"
by auto
ML "Header.record \"LengthEqualCons\""

theorem ZipSpec :
"ALL xs.
 ALL ys.
 length'(xs) = length'(ys) --> unzip(X_zip xs ys) = (xs, ys)"
oops
ML "Header.record \"ZipSpec\""

end
```

B.9 Prelude_Char.thy

```
theorem ICE02 : "ALL x. ALL y. Not' (ord'(x) ==' ord'(y))
 = x /= y"
apply(auto)
apply(simp add: DiffDef)
done
ML "Header.record \"ICE02\""

theorem ICO05 : "ALL x. ALL y. ord'(x) <='' ord'(y)
 = x <='' y"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"ICO05\""

theorem ICO06 : "ALL x. ALL y. ord'(x) >'' ord'(y)
 = x >'' y"
apply(rule allI)+
apply(simp add: GeDef)
done
ML "Header.record \"ICO06\""

theorem ICO07 : "ALL x. ALL y. ord'(x) >='' ord'(y)
 = x >='' y"
apply(rule allI)+
```

```
apply(simp only: GeqDef)
apply(simp add: GeDef)
done
ML "Header.record \"IC007\""

theorem IC001 :
"ALL x. ALL y. compare x y ==' EQ = ord'(x) ==' ord'(y)"
by auto
ML "Header.record \"IC001\""

theorem IC002 :
"ALL x. ALL y. compare x y ==' LT = ord'(x) <'' ord'(y)"
by auto
ML "Header.record \"IC002\""

theorem IC003 :
"ALL x. ALL y. compare x y ==' GT = ord'(x) >'' ord'(y)"
apply(rule allI)+
apply(simp add: GeDef)
done
ML "Header.record \"IC003\""

theorem IC008 :
"ALL x. ALL y. ord'(x) <='' ord'(y) = X_maxX2 x y ==' y"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IC008\""

theorem IC009 :
"ALL x. ALL y. ord'(y) <='' ord'(x) = X_maxX2 x y ==' x"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IC009\""

theorem IC010 :
"ALL x. ALL y. ord'(x) <='' ord'(y) = X_minX2 x y ==' x"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IC010\""

theorem IC011 :
"ALL x. ALL y. ord'(y) <='' ord'(x) = X_minX2 x y ==' y"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IC011\""

end
```

B.10 Prelude_String.thy

```
theorem StringT1 :
"ALL x.
 ALL xs.
 ALL y. x ==' y = True'
 --> X_Cons x xs ==' X_Cons y xs = True'"
apply(auto)
apply(simp add: ILE02)
done
ML "Header.record \"StringT1\""

theorem StringT2 :
"ALL x.
 ALL xs.
 ALL y.
 ALL ys. xs /= ys = True'
--> X_Cons x ys ==' X_Cons y xs = False'"
apply(auto)
apply(simp add: ILE02)
apply(case_tac "x ==' y")
apply(auto)
```

```
apply(simp add: EqualSymDef)
apply(simp add: DiffDef)
apply(simp add: NotFalse1)
done
ML "Header.record \"StringT2\""

theorem StringT3 :
"ALL a. ALL b. a /= b = True'
--> a ==' b = False'"
apply(auto)
apply(simp add: DiffDef)
apply(simp add: NotFalse1)
done
ML "Header.record \"StringT3\""

theorem StringT4 :
"ALL x.
 ALL xs.
 ALL y. x <'' y = True'
 --> X_Cons x xs <'' X_Cons y xs = True'"
by auto
ML "Header.record \"StringT4\""

theorem StringT5 :
"ALL x.
 ALL y.
 ALL z.
 x <'' y = True' & y <'' z = True' -->
 X_Cons x (X_Cons z Nil') <'' X_Cons x (X_Cons y Nil')
 = False'"
by auto
ML "Header.record \"StringT5\""

end
```

B.11 Prelude_ExamplePrograms_E1.thy

```
theorem Program01 :
"andL(X_Cons True' (X_Cons True' (X_Cons True' Nil')))
= True'"
apply(simp only: AndLDef)
apply(simp only: FoldrCons)
apply(simp only: FoldrNil)
apply(simp add: AndPrefixDef)
done
ML "Header.record \"Program01\""

theorem Program02 :
"quickSort(X_Cons True' (X_Cons False' Nil')) =
 X_Cons False' (X_Cons True' Nil')"
apply(simp only: QuickSortCons)
apply(case_tac "(%y. y <'' True') False'")
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusNil)
apply(simp only: XPlusXPlusCons)
apply(simp only: XPlusXPlusNil)
apply(case_tac "(%y. y >='' True') False'")
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp add: LeFGeTEqTRel)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortCons)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusNil)
apply(simp only: XPlusXPlusCons)
apply(simp only: XPlusXPlusNil)
apply(simp only: IBO5)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortCons)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusNil)
apply(simp only: XPlusXPlusCons)
```

```
apply(simp only: XPlusXPlusNil)
apply(case_tac "(%y. y >='' True') False'")
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusCons)
apply(simp only: XPlusXPlusNil)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortCons)
apply(simp only: FilterNil FilterConsT FilterConsF)
apply(simp only: QuickSortNil)
apply(simp only: XPlusXPlusNil)
apply(simp only: XPlusXPlusCons)
apply(simp only: XPlusXPlusNil)
apply(simp add: LeFGeTEqTRel)
done
ML "Header.record \"Program02\""

theorem Program03 :
"insertionSort(X_Cons True' (X_Cons False' Nil')) =
 X_Cons False' (X_Cons True' Nil')"
apply(simp only: InsertionSortConsCons)
apply(simp only: InsertionSortNil)
apply(simp only: InsertNil)
apply(case_tac "True' >'' False'")
apply(simp only: GeFLeTEqTRel)
apply(simp add: LeqTLeTEqTRel)
apply(simp only: InsertCons2)
apply(simp only: InsertNil)
done
ML "Header.record \"Program03\""

theorem Program04 : "ALL xs. insertionSort(xs)
 = quickSort(xs)"
oops
ML "Header.record \"Program04\""
end
```

Apêndice C

Listagem das Especificações com Avaliação Preguiçosa Desenvolvidas em HasCASL

Este apêndice contém o código das especificações com avaliação preguiçosa desenvolvidas neste trabalho com o uso da linguagem HasCasl. O arquivo-fonte pode ser reconstruído compiando-se todas as especificações aqui descritas, na ordem apresentada, em um arquivo *Prelude.hs*.

C.1 Cabeçalhos da Biblioteca *Prelude*

```
library LazyPrelude

version 0.1

%authors: Glauber M. Cabral <glauber.sp@gmail.com>

%date: 01 Jun 2009


logic HasCASL
```

```
from Basic/Numbers get Nat, Int, Rat
from HasCASL/Metatheory/Monad get Functor, Monad
from Basic/CharactersAndStrings get Char |-> IChar
```

C.2 Especificação *Bool*

```
spec Bool = %mono
type Bool := ?Unit
fun Not__ : ?Bool -> ?Bool
fun __&&__ : ?Bool * ?Bool -> ?Bool
fun __||__ : ?Bool * ?Bool -> ?Bool
fun otherwiseH: ?Bool
vars x,y: ?Bool
. Not(false)                        %(NotFalse)%
. not Not(true)                     %(NotTrue)%
. not (false && x)                  %(AndFalse)%
. true && x = x                     %(AndTrue)%
. x && y = y && x                   %(AndSym)%
. x || y = Not(Not(x) && Not(y))    %(OrDef)%
. otherwiseH                        %(OtherwiseDef)%
. (Not x) = not x           %(NotFalse1)% %implied
. not (Not x) = x           %(NotTrue1)% %implied
. (not x) = Not x           %(notNot1)% %implied
. (not not x) = not Not x   %(notNot2)% %implied
end
```

C.3 Especificação *Eq*

```
spec Eq = Bool then
class Eq {
var a: Eq
fun __==__ : ?a * ?a -> ?Bool
fun __/=__ : ?a * ?a -> ?Bool
vars x,y,z: ?a
. x = y => (x == y)                              %(EqualTDef)%
. (x == y) = (y == x)                            %(EqualSymDef)%
. x == x                                         %(EqualReflex)%
. (x == y) /\ (y == z) => (x == z)               %(EqualTransT)%
. (x /= y) = Not (x == y)                        %(DiffDef)%
. (x /= y) = (y /= x)                            %(DiffSymDef)% %implied
. (x /= y) <=> Not (x == y)                      %(DiffTDef)% %implied
. not (x /= y) <=> (x == y)                      %(DiffFDef)% %implied
. not (x == y) => not (x = y)           %(TE1)% %implied
. Not (x == y) <=> not (x == y)         %(TE2)% %implied
. not (Not (x == y)) <=> (x == y)       %(TE3)% %implied
. not (x == y) <=> not (x == y)         %(TE4)% %implied
}
type instance Unit: Eq
. (() == ())                            %(IUE1)% %implied
. not (() /= ())                        %(IUE2)% %implied
type instance Bool: Eq
. (true == true)                        %(IBE1)% %implied
. (false == false)                      %(IBE2)% %implied
. not (false == true)                   %(IBE3)%
. not (true == false)                   %(IBE4)% %implied
. (true /= false)                       %(IBE5)% %implied
. (false /= true)                       %(IBE6)% %implied
```

```
. Not (true == false)                         %(IBE7)% %implied
. not (Not (Not (true == false)))             %(IBE8)% %implied
end
```

C.4 Especificação *Ord*

```
spec Ord = Eq and Bool then
free type Ordering ::= LT | EQ | GT
type instance Ordering: Eq
. (LT == LT)          %(IOE01)% %implied
. (EQ == EQ)          %(IOE02)% %implied
. (GT == GT)          %(IOE03)% %implied
. not (LT == EQ)      %(IOE04)%
. not (LT == GT)      %(IOE05)%
. not (EQ == GT)      %(IOE06)%
. (LT /= EQ)          %(IOE07)% %implied
. (LT /= GT)          %(IOE08)% %implied
. (EQ /= GT)          %(IOE09)% %implied
class Ord < Eq
{
 var a: Ord
 fun compare: ?a -> ?a -> ?Ordering
 fun __<__ : ?a * ?a -> ?Bool
 fun __>__ : ?a * ?a -> ?Bool
 fun __<=__ : ?a * ?a -> ?Bool
 fun __>=__ : ?a * ?a -> ?Bool
 fun min: ?a -> ?a -> ?a
 fun max: ?a -> ?a -> ?a
 var    x, y, z, w: ?a
%% Definitions for relational operations.
```

```
%% Axioms for <
 . (x == y) => not (x < y)                      %(LeIrreflexivity)%
%% . not (x < x)                                  %(LeIrreflexivity)%
 . (x < y) => not (y < x)                       %(LeTAsymmetry)% %implied
 . (x < y) /\ (y < z) => (x < z)                %(LeTTransitive)%
 . (x < y) \/ (y < x) \/ (x == y)               %(LeTTotal)%
%% Axioms for >
 . (x > y) = (y < x)                            %(GeDef)%
 . (x == y)  => not (x > y)                     %(GeIrreflexivity)% %implied
 . (x > y)  => not (y > x)                      %(GeTAsymmetry)% %implied
 . ((x > y)  && (y > z))  => (x > z)            %(GeTTransitive)% %implied
 . (((x > y) || (y > x)) || (x == y))           %(GeTTotal)% %implied
%% Axioms for <=
 . (x <= y) = (x < y) || (x == y)               %(LeqDef)%
 . (x <= x)                                     %(LeqReflexivity)% %implied
 . ((x <= y) && (y <= z))  => (x <= z)          %(LeqTTransitive)% %implied
 . (x <= y) && (y <= x) = (x == y)              %(LeqTTotal)% %implied
%% Axioms for >=
 . (x >= y) = ((x > y) || (x == y))             %(GeqDef)%
 . (x >= x)                                     %(GeqReflexivity)% %implied
 . ((x >= y) && (y >= z))  => (x >= z)          %(GeqTTransitive)% %implied
 . (x >= y) && (y >= x) = (x == y)              %(GeqTTotal)% %implied
%% Relates == and ordering
 . (x == y)  <=> not (x < y) /\ not (x > y)     %(EqTSOrdRel)%
 . not (x == y) <=> (x < y)  \/ (x > y)         %(EqFSOrdRel)%
 . (x == y)  <=> (x <= y)  /\ (x >= y)          %(EqTOrdRel)%
 . not (x == y) <=> (x <= y)  \/ (x >= y)       %(EqFOrdRel)%
 . (x == y)  /\ (y < z)  => (x < z)             %(EqTOrdTSubstE)%
 . (x == y)  /\ not (y < z) => not (x < z)      %(EqTOrdFSubstE)%
 . (x == y)  /\ (z < y)  => (z < x)             %(EqTOrdTSubstD)%
 . (x == y)  /\ not (z < y) => not (z < x)      %(EqTOrdFSubstD)%
```

```
 . (x < y)  <=> not (x > y) /\ not (x == y)      %(LeTGeFEqFRel)%
 . not (x < y) <=> (x > y)  \/ (x == y)          %(LeFGeTEqTRel)%
%% Relates all the ordering operators with true as result.
 . (x < y) <=> (y > x)                           %(LeTGeTRel)% %implied
 . not (x < y) <=> not (y > x)                   %(LeFGeFRel)% %implied
 . (x <= y) <=> (y >= x)                         %(LeqTGetTRel)% %implied
 . not (x <= y) <=> not (y >= x)                 %(LeqFGetFRel)% %implied
 . (x > y)  <=> (y < x)                          %(GeTLeTRel)% %implied
 . not (x > y) <=> not (y < x)                   %(GeFLeFRel)% %implied
 . (x >= y) <=> (y <= x)                         %(GeqTLeqTRel)% %implied
 . not (x >= y) <=> not (y <= x)                 %(GeqFLeqFRel)% %implied
%%
 . (x <= y) <=> not (x > y)                      %(LeqTGeFRel)% %implied
 . not (x <= y) <=> (x > y)                      %(LeqFGeTRel)% %implied
 . (x > y) <=> not (x < y) /\ not (x == y)       %(GeTLeFEqFRel)% %implied
 . not (x > y) <=> (x < y)  \/ (x == y)          %(GeFLeTEqTRel)% %implied
 . (x >= y) <=> not (x < y)                      %(GeqTLeFRel)% %implied
 . not (x >= y) <=> (x < y)                      %(GeqFLeTRel)% %implied
%%
 . (x <= y) <=> (x < y)  \/ (x == y)             %(LeqTLeTEqTRel)% %implied
 . not (x <= y) <=> not (x < y) /\ not (x == y)  %(LeqFLeFEqFRel)% %implied
 . (x >= y) <=> (x > y)  \/ (x == y)             %(GeqTGeTEqTRel)% %implied
 . not (x >= y) <=> not (x > y) /\ not (x == y)  %(GeqFGeFEqFRel)% %implied
%% Implied true - false relations.
 . (x < y)  <=> not (x >= y)                     %(LeTGeqFRel)% %implied
 . (x > y)  <=> not (x <= y)                     %(GeTLeqFRel)% %implied
 . (x < y) = (x <= y) && (x /= y)                %(LeLeqDiff)% %implied
%% Definitions to compare, max and min using relational operations.
 . (compare x y == LT) = (x < y)                     %(CmpLTDef)%
 . (compare x y == EQ) = (x == y)                    %(CmpEQDef)%
 . (compare x y == GT) = (x > y)                     %(CmpGTDef)%
```

```
%% Define min, max
 . (max x y == y) = (x <= y)                          %(MaxYDef)%
 . (max x y == x) = (y <= x)                          %(MaxXDef)%
 . (min x y == x) = (x <= y)                          %(MinXDef)%
 . (min x y == y) = (y <= x)                          %(MinYDef)%
 . (max x y == y) = (max y x == y)                    %(MaxSym)% %implied
 . (min x y == y) = (min y x == y)                    %(MinSym)% %implied
}
 . (x == y) \/ (x < y) <=> (x <= y)                   %(TO1)% %implied
 . (x == y) => not (x < y)                            %(TO2)% %implied
 . Not (Not (x < y))  \/ Not (x < y)                  %(TO3)% %implied
 . (x < y)  => Not (x == y)                           %(TO4)% %implied
 . (x < y)  /\ (y < z)  /\ (z < w)  => (x < w)        %(TO5)% %implied
 . (z < x)  => Not (x < z)                            %(TO6)% %implied
 . (x < y)  <=> (y > x)                               %(TO7)% %implied
 type instance Unit: Ord
 . (() <= ())                     %(IUO01)% %implied
 . not (() <  ())                 %(IUO02)% %implied
 . (() >= ())                     %(IUO03)% %implied
 . not (() > ())                  %(IUO04)% %implied
 . (max () () == ())              %(IUO05)% %implied
 . (min () () == ())              %(IUO06)% %implied
 . (compare () () == EQ)          %(IUO07)% %implied
 type instance Ordering: Ord
 . (LT < EQ)                      %(IO013)%
 . (EQ < GT)                      %(IO014)%
 . (LT < GT)                      %(IO015)%
 . (LT <= EQ)                     %(IO016)% %implied
 . (EQ <= GT)                     %(IO017)% %implied
 . (LT <= GT)                     %(IO018)% %implied
 . (EQ >= LT)                     %(IO019)% %implied
```

```
 . (GT >= EQ)                     %(IO020)% %implied
 . (GT >= LT)                     %(IO021)% %implied
 . (EQ > LT)                      %(IO022)% %implied
 . (GT > EQ)                      %(IO023)% %implied
 . (GT > LT)                      %(IO024)% %implied
 . (max LT EQ == EQ)              %(IO025)% %implied
 . (max EQ GT == GT)              %(IO026)% %implied
 . (max LT GT == GT)              %(IO027)% %implied
 . (min LT EQ == LT)              %(IO028)% %implied
 . (min EQ GT == EQ)              %(IO029)% %implied
 . (min LT GT == LT)              %(IO030)% %implied
 . (compare LT LT == EQ)          %(IO031)% %implied
 . (compare EQ EQ == EQ)          %(IO032)% %implied
 . (compare GT GT == EQ)          %(IO033)% %implied
 type instance Bool: Ord
 . (false < true)                 %(IB05)%
 . not (false >= true)            %(IB06)% %implied
 . (true >= false)                %(IB07)% %implied
 . not (true < false)             %(IB08)% %implied
 . (max false true == true)       %(IB09)% %implied
 . (min false true == false)      %(IB010)% %implied
 . (compare true true == EQ)      %(IB011)% %implied
 . (compare false false == EQ)    %(IB012)% %implied
 type instance Nat: Ord
end
```

C.5 Especificação *Maybe*

```
spec Maybe = Eq and Ord then
var a,b,c : Type;
```

```
    e : Eq;
    o : Ord;
free type Maybe a ::= Just (?a)? | Nothing
var x : ?a;
    y : ?b;
    ma : ?Maybe a;
    f : ?a -> ?b
fun maybe : ?b -> (?a -> ?b) -> ?Maybe a -> ?b
. maybe y f (Just x: ?Maybe a) = f x                   %(MaybeJustDef)%
. maybe y f (Nothing: Maybe a) = y                     %(MaybeNothingDef)%
type instance Maybe e: Eq
var x,y : ?e;
. (Just x == Just y)  <=> (x == y)                     %(IME01)%
. ((Nothing : Maybe e) == (Nothing: Maybe e))          %(IME02)% %implied
. not Just x == Nothing                                %(IME03)%
type instance Maybe o: Ord
var x,y : ?o;
. (Nothing < Just x)                                   %(IMO01)%
. (Just x < Just y) = (x < y)                          %(IMO02)%
. not (Nothing >= Just x)                              %(IMO03)% %implied
. (Just x >= Nothing)                                  %(IMO04)% %implied
. not (Just x < Nothing)                               %(IMO05)% %implied
. (compare Nothing (Just x) == EQ)
     = (Nothing == (Just x))                           %(IMO06)% %implied
. (compare Nothing (Just x) == LT)
     = (Nothing < (Just x))                            %(IMO07)% %implied
. (compare Nothing (Just x) == GT)
     = (Nothing > (Just x))                            %(IMO08)% %implied
. (Nothing <= (Just x))
     = (max Nothing (Just x) == (Just x))              %(IMO09)% %implied
. ((Just x) <= Nothing)
```

```
     = (max Nothing (Just x) == Nothing)              %(IMO10)% %implied
. (Nothing <= (Just x))
     = (min Nothing (Just x) == Nothing)              %(IMO11)% %implied
. ((Just x) <= Nothing)
     = (min Nothing (Just x) == (Just x))             %(IMO12)% %implied
end
```

C.6 Especificação *MaybeMonad*

```
spec MaybeMonad = Maybe and Monad then
var a,b,c : Type;
type instance Maybe: Functor
vars  x: ?Maybe a;
      f: ?a -> ?b;
      g: ?b -> ?c
. map (\ y: a .! y) x = x                           %(IMF01)% %implied
. map (\ y: a .! g (f y)) x = map g (map f x)       %(IMF02)% %implied
type instance Maybe: Monad
vars  x, y: ?a;
      p: ?Maybe a;
      q: ?a -> ?Maybe b;
      r: ?b -> ?Maybe c;
      f: ?a -> ?b
. def q x => ret x >>= q = q x                      %(IMM01)% %implied
. p >>= (\ x: a . ret (f x) >>= r)
     = p >>= \ x: a . r (f x)                       %(IMM02)% %implied
. p >>= ret = p                                     %(IMM03)% %implied
. (p >>= q) >>= r = p >>= \ x: a . q x >>= r        %(IMM04)% %implied
. (ret x : Maybe a) = ret y => x = y                %(IMM05)% %implied
var x : ?Maybe a;
```

```
    f : ?a -> ?b;
. map f x = x >>= (\ y:a . ret (f y))                  %(T01)% %implied
end
```

C.7 Especificação *Either*

```
spec Either = Eq and Ord then
var a, b, c : Type; e, ee : Eq; o, oo : Ord;
free type Either a b ::= Left (?a)? | Right (?b)?
var x : ?a; y : ?b; z : ?c; eab : ?Either a b; f : ?a -> ?c; g : ?b -> ?c
fun either : (?a -> ?c) -> (?b -> ?c) -> ?Either a b -> ?c
. either f g (Left x: ?Either a b) = f x                  %(EitherLeftDef)%
. either f g (Right y: ?Either a b) = g y                 %(EitherRightDef)%
type instance Either e ee: Eq
var x,y : ?e; z,w : ?ee;
. ((Left x : ?Either e ee) ==
   (Left y : ?Either e ee)) = (x == y)                    %(IEE01)%
. ((Right z : ?Either e ee) ==
   (Right w : ?Either e ee)) = (z == w)                   %(IEE02)%
. not ((Left x : ?Either e ee) ==
   (Right z : ?Either e ee))                              %(IEE03)%
type instance Either o oo: Ord
var x,y : ?o; z,w : ?oo;
. ((Left x : ?Either o oo) < (Right z : ?Either o oo))
                                                          %(IEO01)%
. ((Left x : ?Either o oo) < (Left y : ?Either o oo))
     = (x < y)                                            %(IEO02)%
. ((Right z : ?Either o oo) < (Right w : ?Either o oo))
     = (z < w)                                            %(IEO03)%
. not ((Left x : ?Either o oo) >= (Right z : ?Either o oo)) %(IEO04)% %implied
```

```
. ((Right z : ?Either o oo) >= (Left x : ?Either o oo))     %(IEO05)% %implied
. not ((Right z : ?Either o oo) < (Left x : ?Either o oo))  %(IEO06)% %implied
. (compare (Left x : ?Either o oo) (Right z : ?Either o oo) == EQ)
     = ((Left x) == (Right z))                             %(IEO07)% %implied
. (compare (Left x : ?Either o oo) (Right z : ?Either o oo) == LT)
     = ((Left x) < (Right z))                              %(IEO08)% %implied
. (compare (Left x : ?Either o oo) (Right z : ?Either o oo) == GT)
     = ((Left x) > (Right z))                              %(IEO09)% %implied
. ((Left x : ?Either o oo) <= (Right z : ?Either o oo))
     = (max (Left x) (Right z) == (Right z))               %(IEO10)% %implied
. ((Right z : ?Either o oo) <= (Left x : ?Either o oo))
     = (max (Left x) (Right z) == (Left x))                %(IEO11)% %implied
. ((Left x : ?Either o oo) <= (Right z : ?Either o oo))
     = (min (Left x) (Right z) == (Left x))                %(IEO12)% %implied
. ((Right z : ?Either o oo) <= (Left x : ?Either o oo))
     = (min (Left x) (Right z) == (Right z))               %(IEO13)% %implied
end
```

C.8 Especificação *EitherFunctor*

```
spec EitherFunctor = Either and Functor then
var a, b, c : Type;
type instance Either a: Functor
vars x: Either c a;
     f: a -> b;
     g: b -> c
. map (\ y: a .! y) x = x                        %(IEF01)% %implied
. map (\ y: a .! g (f y)) x = map g (map f x)    %(IEF02)% %implied
end
```

C.9 Especificação *Composition*

```
spec Composition =
vars a,b,c : Type
fun __o__ : (?b -> ?c) * (?a -> ?b) -> (?a -> ?c);
vars a,b,c : Type; y: ?a;
     f : ?b -> ?c;
     g : ?a -> ?b
. ((f o g) y) = f (g y)                 %(Comp1)%
end
```

C.10 Especificação *Function*

```
spec Function = Composition then
var a,b,c: Type;
    x: ?a;
    y: ?b;
    f: ?a -> ?b -> ?c;
    g: (?a * ?b) -> ?c
fun id: ?a -> ?a
fun flip: (?a -> ?b -> ?c) -> ?b -> ?a -> ?c
fun fst: (?a * ?b) -> ?a
fun snd: (?a * ?b) -> ?b
fun curry: ((?a * ?b) -> ?c) -> ?a -> ?b -> ?c
fun uncurry: (?a -> ?b -> ?c) -> (?a * ?b) -> ?c
. id x = x                      %(IdDef)%
. flip f y x = f x y            %(FlipDef)%
. fst (x, y) = x                %(FstDef)%
. snd (x, y) = y                %(SndDef)%
. curry g x y = g (x, y)        %(CurryDef)%
. uncurry f (x,y) = f x y       %(UncurryDef)%
```

```
end
```

C.11 Especificação *ListNoNumbers*

```
spec ListNoNumbers = Function and Ord then
var a : Type
free type List a ::= Nil | Cons (?a) (?List a)?
fun Cons: forall a: Type. (?a) --> (?List a) --> (?List a)
type DList a := List (List a)
var a,b : Type
fun head : ?List a -> ?a;
fun tail : ?List a -> ?List a;
fun foldr : (?a -> ?b -> ?b) -> ?b -> ?List a -> ?b;
fun foldl : (?a -> ?b -> ?a) -> ?a -> ?List b -> ?a;
fun map : (?a -> ?b) -> ?List a -> ?List b;
fun filter : (?a -> ?Bool) -> ?List a -> ?List a;
fun __++__ : ?List a * ?List a -> ?List a;
fun zip : ?List a -> ?List b -> ?List (?a * ?b);
fun unzip : ?List (?a * ?b) -> ?(?List a * ?List b)
vars a,b : Type;
     f : ?a -> ?b -> ?b;
     g : ?a -> ?b -> ?a;
     h : ?a -> ?b;
     p : ?a -> ?Bool;
     x,y,t : ?a;
     xs,ys,l : ?List a;
     z,s : ?b;
     zs : ?List b;
     ps : ?List (?a * ?b)
. not def head (Nil : List a)                               %(NotDefHead)%
```

```
. head (Cons x xs) = x                                       %(HeadDef)%
. not def tail (Nil : List a)                                %(NotDefTail)%
. tail (Cons x xs) = xs                                      %(TailDef)%
. foldr f s Nil = s                                          %(FoldrNil)%
. foldr f s (Cons x xs)
    = f x (foldr f s xs)                                     %(FoldrCons)%
. foldl g t Nil = t                                          %(FoldlNil)%
. foldl g t (Cons z zs)
    = foldl g (g t z) zs                                     %(FoldlCons)%
. map h Nil = Nil                                            %(MapNil)%
. map h (Cons x xs)
    = (Cons (h x) (map h xs))                                %(MapCons)%
. Nil ++ l = l                                               %(++Nil)%
. (Cons x xs) ++ l = Cons x (xs ++ l)                        %(++Cons)%
. filter p Nil = Nil                                         %(FilterNil)%
. p x
    => filter p (Cons x xs) = Cons x (filter p xs)           %(FilterConsT)%
. not p x
    => filter p (Cons x xs) = filter p xs                    %(FilterConsF)%
. zip (Nil : List ?a) l = Nil                                %(ZipNil)%
. l = Nil
    => zip (Cons x xs) l = Nil                               %(ZipConsNil)%
. l = (Cons y ys)
    => zip (Cons x xs) l = Cons (x,y) (zip xs ys)            %(ZipConsCons)%
      . unzip (Nil : ?List (?a * ?b))
          = ((Nil: ?List (?a)), (Nil: ?List (?b)))               %(UnzipNil)%
      . (ys, zs) = unzip ps
          => unzip (Cons (x,z) ps)
               = (Cons x ys, Cons z zs)                          %(UnzipCons)%

then
```

```
var a : Eq; x,y: ?a; xs, ys: ?List a
type instance List a: Eq
. ((Nil: List a) == (Nil: List a))                          %(ILE01)% %implied
. ((Cons x xs) == (Cons y ys)) = ((x == y) && (xs == ys))   %(ILE02)%
var b : Ord; z,w: ?b; zs, ws: ?List b
type instance List b: Ord
. not ((Nil: List b) < (Nil: List b))                       %(ILO01)% %implied
. ((Nil: List b) <= (Nil: List b))                          %(ILO02)% %implied
. not ((Nil: List b) > (Nil: List b))                       %(ILO03)% %implied
. ((Nil: List b) >= (Nil: List b))                          %(ILO04)% %implied
. (z < w)  => ((Cons z zs) < (Cons w ws))                   %(ILO05)%
. (z == w)  => ((Cons z zs) < (Cons w ws)) = (zs < ws)      %(ILO06)%
. not (z < w) /\ not (z == w)
     => not ((Cons z zs) < (Cons w ws))                     %(ILO07)%
. ((Cons z zs) <= (Cons w ws))
     = ((Cons z zs) < (Cons w ws))
          || ((Cons z zs) == (Cons w ws))                   %(ILO08)% %implied
. ((Cons z zs) > (Cons w ws))
     = ((Cons w ws) < (Cons z zs))                          %(ILO09)% %implied
. ((Cons z zs) >= (Cons w ws))
     = ((Cons z zs) > (Cons w ws))
          || ((Cons z zs) == (Cons w ws))                   %(ILO10)% %implied
. (compare (Nil: List b) (Nil: List b) == EQ)
     = ((Nil: List b) == (Nil: List b))                     %(ILO11)% %implied
. (compare (Nil: List b) (Nil: List b) == LT)
     = ((Nil: List b) < (Nil: List b))                      %(ILO12)% %implied
. (compare (Nil: List b) (Nil: List b) == GT)
     = ((Nil: List b) > (Nil: List b))                      %(ILO13)% %implied
. (compare (Cons z zs) (Cons w ws) == EQ)
     = ((Cons z zs) == (Cons w ws))                         %(ILO14)% %implied
. (compare (Cons z zs) (Cons w ws) == LT)
```

```
     = ((Cons z zs) < (Cons w ws))                      %(ILO15)% %implied
. (compare (Cons z zs) (Cons w ws) == GT)
     = ((Cons z zs) > (Cons w ws))                      %(ILO16)% %implied
. (max (Nil: List b) (Nil: List b) == (Nil: List b))
     = ((Nil: List b) <= (Nil: List b))                 %(ILO17)% %implied
. (min (Nil: List b) (Nil: List b) == (Nil: List b))
     = ((Nil: List b) <= (Nil: List b))                 %(ILO18)% %implied
. ((Cons z zs) <= (Cons w ws))
     = (max (Cons z zs) (Cons w ws) == (Cons w ws))     %(ILO19)% %implied
. ((Cons w ws) <= (Cons z zs))
     = (max (Cons z zs) (Cons w ws) == (Cons z zs))     %(ILO20)% %implied
. ((Cons z zs) <= (Cons w ws))
     = (min (Cons z zs) (Cons w ws) == (Cons z zs))     %(ILO21)% %implied
. ((Cons w ws) <= (Cons z zs))
     = (min (Cons z zs) (Cons w ws) == (Cons w ws))     %(ILO22)% %implied
then %implies
vars a,b,c : Ord;
     f : ?a -> ?b;
     g : ?b -> ?c;
     h : ?a -> ?a -> ?a;
     i : ?a -> ?b -> ?a;
     p : ?b -> ?Bool;
     x: ?a;
     y: ?b;
     xs,zs : ?List a;
     ys,ts : ?List b;
     z,e : ?a;
     xxs : ?List (List a)
. foldl i e (ys ++ ts)
     = foldl i (foldl i e ys) ts                          %(FoldlDecomp)%
. map f (xs ++ zs)
```

```
     = (map f xs) ++ (map f zs)                                  %(MapDecomp)%
. map (g o f) xs = map g (map f xs)                              %(MapFunctor)%
. filter p (map f xs)
     = map f (filter (p o f) xs)                                 %(FilterProm)%
then
vars a,b: Type;
     x,q,r: ?a;
     xs,qs,rs: ?List a;
     y,z: ?b;
     ys,zs: ?List b;
     f: ?a -> ?a -> ?a;
     g: ?a -> ?a -> ?a;
     h: ?a -> ?a -> ?a;
fun init: ?List a -> ?List a;
fun last: ?List a -> ?a;
fun null: ?List a -> ?Bool;
fun reverse: ?List a -> ?List a;
fun foldr1: (?a -> ?a -> ?a) -> ?List a -> ?a;
fun foldl1: (?a -> ?a -> ?a) -> ?List a -> ?a;
fun scanl: (?a -> ?b -> ?a) -> ?a -> ?List b -> ?List a
fun scanl1: (?a -> ?a -> ?a) -> ?List a -> ?List a
fun scanr: (?a -> ?b -> ?b) -> ?b -> ?List a -> ?List b
fun scanr1: (?a -> ?a -> ?a) -> ?List a -> ?List a
. not def init (Nil: List a)                                     %(InitNil)%
. init (Cons x (Nil: List a)) = (Nil:List a)                     %(InitConsNil)%
. init (Cons x xs) = Cons x (init xs)                            %(InitConsCons)%
. not def last (Nil: List a)                                     %(LastNil)%
. last (Cons x (Nil: List a)) = x                                %(LastConsNil)%
. last (Cons x xs) = last xs                                     %(LastConsCons)%
. null (Nil:List a)                                              %(NullNil)%
. not null (Cons x xs)                                           %(NullCons)%
```

```
. reverse (Nil: List a) = (Nil: List a)                              %(ReverseNil)%
. reverse (Cons x xs) = (reverse xs) ++ (Cons x (Nil: List a)) %(ReverseCons)%
. not def foldr1 f (Nil: List a)                                     %(Foldr1Nil)%
. foldr1 f (Cons x (Nil: List a)) = x                                %(Foldr1ConsNil)%
. foldr1 f (Cons x xs) = f x (foldr1 f xs)                           %(Foldr1ConsCons)%
. not def foldl1 f (Nil: List a)                                     %(Foldl1Nil)%
. foldl1 f (Cons x (Nil: List a)) = x                                %(Foldl1ConsNil)%
. foldl1 f (Cons x xs) = f x (foldr1 f xs)                           %(Foldl1ConsCons)%
. xs = (Nil: List a) => scanl g q xs = Cons q (Nil: List a)            %(ScanlNil)%
. xs = (Cons r rs) => scanl g q xs = Cons q (scanl g (g q r) rs)  %(ScanlCons)%
. scanl1 f Nil = Nil                                                   %(Scanl1Nil)%
. scanl1 f (Cons x xs) = scanl f x xs                                  %(Scanl1Cons)%
. scanr h q (Nil: List a) = Cons q (Nil: List a)                       %(ScanrNil)%
. (Cons r rs) = scanr h q xs
   => scanr h q (Cons x xs) = Cons (h x r)  (Cons r rs)                %(ScanrCons)%
. scanr1 f (Nil:List a) = (Nil:List a)                                 %(Scanr1Nil)%
. scanr1 f (Cons x (Nil:List a)) = (Cons x (Nil:List a))               %(Scanr1ConsNil)%
. Cons q qs = scanr1 f xs
   => scanr1 f (Cons x xs) =  Cons (f x q) (Cons q qs)                 %(Scanr1ConsCons)%
. last (scanl g x xs) = foldl g x xs                                   %(ScanlProperty)% %implied
. head (scanr h x xs) = foldr h x xs                                   %(ScanrProperty)% %implied
then
vars a,b,c : Type;
   d : Ord;
   b1,b2: ?Bool;
   x, y : ?a;
   xs, ys, zs : ?List a;
   xxs : ?List (List a);
   r, s : ?d;
   ds : ?List d;
   bs : ?List Bool;
```

```
     f : ?a -> ?a -> ?a;
     p, q : ?a -> ?Bool;
     g : ?a -> ?List b;
     n,nx: Nat;
fun concatMap : (?a -> ?List b) -> ?List a -> ?List b;
fun concat : ?List (List a) -> ?List a;
fun maximum : ?List d -> ?d;
fun minimum : ?List d -> ?d;
fun takeWhile : (?a -> ?Bool) -> ?List a -> ?List a
fun dropWhile  : (?a -> ?Bool) -> ?List a -> ?List a
fun span : (?a -> ?Bool) -> ?List a -> (?List a * ?List a)
fun break : (?a -> ?Bool) -> ?List a -> (?List a * ?List a)
. concat xxs = foldr (curry __++__) (Nil: List a) xxs      %(ConcatDef)%
. concatMap g xs = concat (map g xs)                       %(ConcatMapDef)%
. not def maximum (Nil: List d)                            %(MaximunNil)%
. maximum ds = foldl1 max ds                               %(MaximumDef)%
. not def minimum (Nil: List d)                            %(MinimunNil)%
. minimum ds = foldl1 min ds                               %(MinimumDef)%
. takeWhile p (Nil: List a) = Nil: List a                  %(TakeWhileNil)%
. p x  => takeWhile p (Cons x xs)
     = Cons x (takeWhile p xs)                             %(TakeWhileConsT)%
. not p x => takeWhile p (Cons x xs) = Nil: List a         %(TakeWhileConsF)%
. dropWhile p (Nil: List a) = Nil: List a                  %(DropWhileNil)%
. p x  => dropWhile p (Cons x xs) = dropWhile p xs         %(DropWhileConsT)%
. not p x => dropWhile p (Cons x xs) = Cons x xs           %(DropWhileConsF)%
. span p (Nil: List a) = ((Nil: List a), (Nil: List a))    %(SpanNil)%
. p x  => span p (Cons x xs)
     = let (ys, zs) = span p xs in
          ((Cons x ys), zs)                                %(SpanConsT)%
. not p x => span p (Cons x xs)
     = let (ys, zs) = span p xs in
```

```
          ((Nil: List a), (Cons x xs))                  %(SpanConsF)%
. span p xs = (takeWhile p xs, dropWhile p xs)          %(SpanThm)% %implied
. break p xs = let q = (Not__ o p) in span q xs         %(BreakDef)%
. break p xs = span (Not__ o p) xs                      %(BreakThm)% %implied
then
vars a,b,c : Type;
     d : Ord;
     e: Eq;
     x, y : ?a;
     xs, ys : ?List a;
     q, r : ?d;
     qs, rs : ?List d;
     s,t: ?e;
     ss,ts: ?List e;
     p: ?a -> ?Bool
fun insert: ?d -> ?List d -> ?List d
fun delete: ?e -> ?List e -> ?List e
fun select: (?a -> ?Bool) -> ?a -> (?List a * ?List a) -> (?List a * ?List a)
fun partition: (?a -> ?Bool) -> ?List a -> (?List a * ?List a)
. insert q (Nil: List d) = Cons q Nil                     %(InsertNil)%
. (q <= r)  => insert q (Cons r rs)
     = (Cons q (Cons r rs))                               %(InsertCons1)%
. (q > r)  => insert q (Cons r rs)
     = (Cons r (insert q rs))                             %(InsertCons2)%
. delete s (Nil: List e) = Nil                            %(DeleteNil)%
. (s == t)  => delete s (Cons t ts) = ts                  %(DeleteConsT)%
. not (s == t) => delete s (Cons t ts)
     = (Cons t (delete s ts))                             %(DeleteConsF)%
. (p x)  => select p x (xs, ys) = ((Cons x xs), ys)       %(SelectT)%
. not (p x) => select p x (xs, ys) = (xs, (Cons x ys))    %(SelectF)%
. partition p xs = foldr (select p) ((Nil: List a),(Nil)) xs %(Partition)%
```

```
. partition p xs
    = (filter p xs, filter (Not__ o p) xs)                    %(PartitionProp)% %implied
end
```

C.12 Especificação *NumericClasses*

```
spec NumericClasses = Ord and Nat and Int and Rat then
type instance Pos: Eq
type instance Pos: Ord
type instance Nat: Eq
type instance Nat: Ord
type instance Int: Eq
type instance Int: Ord
type instance Rat: Eq
type instance Rat: Ord
class Num < Eq {
 vars a: Num;
 fun __+__: ?a * ?a -> ?a
 fun __*__: ?a * ?a -> ?a
 fun __-__: ?a * ?a -> ?a
 fun negate: ?a -> ?a
 fun abs: ?a -> ?a
 fun signum: ?a -> ?a
 fun fromInteger: Int -> ?a
}
vars a: Num;
     x,y : ?a
. (abs x) * (signum x) = x                  %(AbsSignumLaw)% %implied
type instance Pos: Num
vars a: Num;
```

```
     x,y: Pos;
     z: Int
. x + y = (__+__: Nat * Nat -> Nat) (x,y)    %(IPN01)%
. x * y = (__*__: Nat * Nat -> Nat) (x,y)    %(IPN02)%
. x - y = (__-!__: Nat * Nat -> Nat) (x,y)   %(IPN03)%
. negate x = 0 -! x                          %(IPN04)%
. (fun abs: ?a -> ?a) x = x                  %(IPN05)%
. signum x = 1                               %(IPN06)%
. fromInteger z = z as Pos                   %(IPN07)%
type instance Nat: Num
vars a: Num;
     x,y: Nat;
     z: Int
. x + y = (__+__: Nat * Nat -> Nat) (x,y)    %(INN01)%
. x * y = (__*__: Nat * Nat -> Nat) (x,y)    %(INN02)%
. x - y = (__-!__: Nat * Nat -> Nat) (x,y)   %(INN03)%
. negate x = 0 -! x                          %(INN04)%
. (fun abs: ?a -> ?a) x = x                  %(INN05)%
. signum x = 1                               %(INN06)%
. fromInteger z = z as Nat                   %(INN07)%
type instance Int: Num
vars a: Num;
     x,y: Int
. x + y = (__+__: Int * Int -> Int) (x,y)             %(IIN01)%
. x * y = (__*__: Int * Int -> Int) (x,y)             %(IIN02)%
. x - y = (__-__: Int * Int -> Int) (x,y)             %(IIN03)%
. negate x = 0 - x                                    %(IIN04)%
. (x >= 0) => (fun abs: ?a -> ?a) x = x               %(IIN05)%
. (x < 0) => (fun abs: ?a -> ?a) x = negate x         %(IIN06)%
. (x > 0) => signum x = 1                             %(IIN07)%
. (x == 0) => signum x = 0                            %(IIN07)%
```

```
. (x < 0) => signum x = - 1                              %(IIN08)%
. fromInteger x = x                                      %(IIN09)%
type instance Rat: Num
vars a: Num;
     x,y: Rat;
     z: Int
. x + y = (__+__: Rat * Rat -> Rat) (x,y)                %(IRN01)%
. x * y = (__*__: Rat * Rat -> Rat) (x,y)                %(IRN02)%
. x - y = (__-__: Rat * Rat -> Rat) (x,y)                %(IRN03)%
. negate x = 0 - x                                       %(IRN04)%
. (x >= 0) => (fun abs: ?a -> ?a) x = x                  %(IRN05)%
. (x < 0) => (fun abs: ?a -> ?a) x = negate x            %(IRN06)%
. (x > 0) => signum x = 1                                %(IRN07)%
. (x == 0) => signum x = 0                               %(IRN07)%
. (x < 0) => signum x = - 1                              %(IRN08)%
. fromInteger z = z / 1                                  %(IRN09)%
%% Integral should be subclass of Real and Enum that we haven't created yet
class Integral < Num
{
vars a: Integral;
fun __quot__, __rem__, __div__, __mod__: ?a * ?a -> ?a
fun quotRem, divMod: ?a -> ?a -> (?a * ?a)
fun toInteger: ?a -> Int
}
type instance Nat: Integral
type instance Int: Integral
type instance Rat: Integral
vars a: Integral;
     x,y,z,w,r,s: ?a;
. (z,w) = quotRem x y => x quot y = z                                        %(IRI01)%
. (z,w) = quotRem x y => x rem y = w                                         %(IRI02)%
```

```
. (z,w) = divMod x y => x div y = z                                  %(IRI03)%
. (z,w) = divMod x y => x mod y = w                                  %(IRI04)%
. signum w = negate (signum y) /\ (z,w) = quotRem x y
        => divMod x y = (z - (fromInteger (toInteger (1:Nat))) , w + s)     %(IRI05)%
. not (signum w = negate (signum y)) /\ (z,w) = quotRem x y
        => divMod x y = (z, w)                                              %(IRI06)%
class Fractional < Num
{
vars a: Fractional
fun __/__ : ?a * ?a -> ?a
fun recip: ?a -> ?a
}
type instance Int: Fractional
type instance Rat: Fractional
vars a: Fractional;
     x,y: Int
. recip x = (1 / x)                            %(IRI01)%
. x / y = x * (recip y)                        %(IRI02)%
vars a: Fractional;
     x,y: Rat
. recip x = (1 / x)                            %(IRF01)%
. x / y = x * (recip y)                        %(IRF02)%
end
```

C.13 Especificação *ListWithNumbers*

```
spec ListWithNumbers = ListNoNumbers and NumericClasses then {
vars a,b: Type;
     c,d: Num;
     x,y : ?a;
```

```
     xs,ys : ?List a;
     n,nx : ?Int;
     z,w: ?Int;
     zs,ws: ?List Int
fun length: ?List a -> Int;
fun take: ?Int -> ?List a -> ?List a
fun drop: ?Int -> ?List a -> ?List a
fun splitAt: ?Int -> ?List a -> (?List a * ?List a)
fun sum: ?List c -> ?c
fun sum': ?List c -> ?c -> ?c
fun product: ?List c -> ?c
fun product': ?List c -> ?c -> ?c
. length (Nil : List a) = 0                                    %(LengthNil)%
. length (Cons x xs) = (length xs) + 1                         %(LengthCons)%
. n <= 0 => take n xs = (Nil:List a)                           %(TakeNegative)%
. take n (Nil:List a) = (Nil:List a)                           %(TakeNil)%
. take n (Cons x xs) = Cons x (take (n-1) xs)                  %(TakeCons)%
. n <= 0 => drop n xs = xs                                     %(DropNegative)%
. drop n (Nil:List a) = (Nil:List a)                           %(DropNil)%
. drop n (Cons x xs) =  drop (n-1) xs                          %(DropCons)%
. splitAt n xs =  (take n xs, drop n xs)                       %(SplitAt)%
. sum' (Nil: List Int) z = z                                   %(Sum'Nil)%
. sum' (Cons z zs) w
     = sum' zs ((fun __+__: ?c * ?c -> ?c)(w,z))               %(Sum'Cons)%
. sum zs = sum' zs 0                                           %(SumL)%
. product' (Nil: List Int) z = z                               %(Prod'Nil)%
. product' (Cons z zs) w
     = product' zs  ((fun __*__: ?c * ?c -> ?c)(w,z))          %(Prod'Cons)%
. product zs = product' zs 1                                   %(ProdL)%
then %implies
     vars a,b,c : Ord;
```

```
          f : ?a -> ?b;
          g : ?b -> ?c;
          h : ?a -> ?a -> ?a;
          i : ?a -> ?b -> ?a;
          p : ?b -> ?Bool;
          x: ?a;
          y: ?b;
          xs,zs : ?List a;
          ys,ts : ?List b;
          z,e : ?a;
          xxs : ?List (List a)
     . length (xs) = 0 <=> xs = Nil                              %(LengthNil1)%
     . length (Nil : List a) = length ys
          => ys = (Nil : List b)                                 %(LengthEqualNil)%
     . length (Cons x xs) = length (Cons y ys)
          => length xs = length ys                               %(LengthEqualCons)%
     . length xs = length ys
          => unzip (zip xs ys) = (xs, ys)                        %(ZipSpec)%
} hide sum', product'
end
```

C.14 Especificação *NumericFunctions*

```
spec NumericFunctions = Function and NumericClasses then {
var a: Num;
    b: Integral;
    c: Fractional
fun subtract: ?a -> ?a -> ?a
fun even: ?b -> ?Bool
fun odd: ?b -> Bool
```

```
fun gcd: ?b -> ?b -> ?b

fun lcm: ?b -> ?b -> ?b

fun gcd': ?b -> ?b -> ?b

fun __^__: ?a * ?b -> ?a

fun f: ?a -> ?b -> ?a

fun g: ?a -> ?b -> ?a -> ?a

fun __^^__: ?c * ?b -> ?c

vars a: Num;

     b: Integral;

     c: Fractional;

     x,y: Int;

     z,w: Int;

     r,s: Rat

. subtract x y = y - x                                    %(Subtract)%

. even z = (z rem (fromInteger 2)) == 0                   %(Even)%

. odd z = Not even z                                      %(Odd)%

. not def gcd 0 0                                         %(GgdUndef)%

. gcd z w = gcd' ((fun abs: ?a -> ?a) z)

     ((fun abs: ?a -> ?a) w)                              %(Gcd)%

. gcd' z 0 = z                                            %(Gcd'Zero)%

. gcd' z w = gcd' w (z rem w)                             %(Gcd')%

. lcm z 0 = 0                                             %(LcmVarZero)%

. lcm (toInteger 0) z = 0                                 %(LcmZeroVar)%

. lcm z w = (fun abs: ?a -> ?a)

     ((z quot ((fun gcd: ?b -> ?b -> ?b) z w)) * w)       %(Lcm)%

. (z < 0) => not def(x ^ z)                               %(ExpUndef)%

. (z == 0) => x ^ z = 1                                   %(ExpOne)%

. (even y) => f x z = f (x * x) (z quot 2);               %(AuxF1)%

. (z == 1) => f x z = x;                                  %(AuxF2)%

. not (even y) /\ not (z == 1)

     => f x z = g (x * x) ((y - 1) quot 2) x;             %(AuxF3)%
```

```
. (even y) => g x z w = g (x * x) (z quot 2) w;              %(AuxG1)%
. (y == 1) => g x z w = x * w;                               %(AuxG2)%
. not (even y) /\ not (y == 1)
     => g x z w = g (x * x) ((z - 1) quot 2) (x * w)         %(AuxG3)%
. not (z < 0) /\ not (z == 0) => x ^ z = f x z               %(Exp)%
} hide f,g
end
```

C.15 Especificação *Char*

```
spec Char = IChar and Ord and NumericClasses then
vars x, y: ?Char
type instance Char: Eq
. (ord(x) == ord(y)) = (x == y)                              %(ICE01)%
. Not(ord(x) == ord(y)) = (x /= y)                           %(ICE02)% %implied
type instance Char: Ord
%% Instance definition of <, <=, >, >=
. (ord(x) < ord(y)) = (x < y)                                %(ICO04)%
. (ord(x) <= ord(y)) = (x <= y)                              %(ICO05)% %implied
. (ord(x) > ord(y)) = (x > y)                                %(ICO06)% %implied
. (ord(x) >= ord(y)) = (x >= y)                              %(ICO07)% %implied
%% Instance definition of compare
. (compare x y == EQ) = (ord(x) == ord(y))                   %(ICO01)% %implied
. (compare x y == LT) = (ord(x) < ord(y))                    %(ICO02)% %implied
. (compare x y == GT) = (ord(x) > ord(y))                    %(ICO03)% %implied
%% Instance defintion of min, max
. (ord(x) <= ord(y)) = (max x y == y)                        %(ICO08)% %implied
. (ord(y) <= ord(x)) = (max x y == x)                        %(ICO09)% %implied
. (ord(x) <= ord(y)) = (min x y == x)                        %(ICO10)% %implied
. (ord(y) <= ord(x)) = (min x y == y)                        %(ICO11)% %implied
```

```
end
```

C.16 Especificação *String*

```
spec String = %mono
     ListNoNumbers and Char then
type String := List Char
vars a,b: ?String; x,y,z: ?Char; xs, ys: ?String
. x == y  => ((Cons x xs) == (Cons y xs))          %(StringT1)% %implied
. xs /= ys  => not ((Cons x ys) == (Cons y xs))     %(StringT2)% %implied
. (a /= b)  =>  not (a == b)                        %(StringT3)% %implied
. (x < y)  =>  ((Cons x xs) < (Cons y xs))          %(StringT4)% %implied
. (x < y)  /\ (y < z)  => not ((Cons x (Cons z Nil))
         < (Cons x (Cons y Nil)))                   %(StringT5)% %implied
end
```

C.17 Especificação *MonadicList*

```
spec MonadicList = Monad and ListNoNumbers then
vars a,b: Type;
     m: Monad;
     f: ?a -> ?m b;
     ms: ?List (?m a);
     k: ?m a -> ?m (List a) -> ?m (List a);
     n: ?m a;
     nn: ?m (List a);
     x: ?a;
     xs: ?List a;
fun sequence: ?List (m a) -> ?m (List a)
fun sequenceUnit: ?List (m a) -> ?m Unit
```

```
fun mapM: (?a -> ?m b) -> ?List a -> ?m (List b)
fun mapMUnit: (?a -> ?m b) -> ?List a -> ?m (List Unit)
. sequence ms = let
  k n nn = n >>= \ x:a. (nn >>= \ xs: List a . (ret (Cons x xs))) in
    foldr k (ret (Nil: List a)) ms                        %(SequenceListDef)%
end
```

C.18 Especificação *ExamplePrograms*

```
spec ExamplePrograms = ListNoNumbers then
var a: Ord;
    x,y: ?a;
    xs,ys: ?List a
fun quickSort: ?List (?a) -> ?List (?a)
fun insertionSort: ?List (?a) -> ?List (?a)
. quickSort (Nil: ?List (?a)) = Nil                       %(QuickSortNil)%
. quickSort (Cons x xs)
     = ((quickSort (filter (\ y:?a .! y < x) xs))
        ++ (Cons x Nil))
          ++ (quickSort (filter (\ y:?a .! y >= x) xs))    %(QuickSortCons)%
. insertionSort (Nil: ?List (?a)) = Nil                   %(InsertionSortNil)%
. insertionSort (Cons x xs) = insert x (insertionSort xs)  %(InsertionSortConsCons)%
then %implies
var a: Ord;
    x,y: ?a;
    xs,ys: ?List a
. quickSort (Cons true (Cons false (Nil: List Bool)))
     = Cons false (Cons true Nil)                         %(Program02)%
. insertionSort (Cons true (Cons false (Nil: List Bool)))
     = Cons false (Cons true Nil)                         %(Program03)%
```

```
. insertionSort xs = quickSort xs                          %(Program04)%
end
```

C.19 Especificação *SortingPrograms*

```
spec SortingPrograms = ListWithNumbers then
var a,b : Ord;
free type Split a b ::= Split (?b) (?List (?List (?a)))
var x,y,z,v,w: ?a;
    r,t: ?b;
    xs,ys,zs,vs,ws: ?List a;
    rs,ts: ?List b;
    xxs: ?List (List a);
    split: ?List a -> ?Split a b;
    join: ?Split a b -> ?List a;
    n: Nat
fun genSort: (?List a -> ?Split a b) -> (?Split a b -> ?List a) -> ?List a -> ?List a
fun splitInsertionSort: ?List b -> ?Split b b
fun joinInsertionSort: ?Split a a -> ?List a
fun insertionSort: ?List a -> ?List a
fun splitQuickSort: ?List a -> ?Split a a
fun joinQuickSort: ?Split b b -> ?List b
fun quickSort: ?List a -> ?List a
fun splitSelectionSort: ?List a -> ?Split a a
fun joinSelectionSort: ?Split b b -> ?List b
fun selectionSort: ?List a -> ?List a
fun splitMergeSort: ?List b -> ?Split b Unit
fun joinMergeSort: ?Split a Unit -> ?List a
fun merge: ?List a -> ?List a -> ?List a
fun mergeSort: ?List a -> ?List a
```

```
. xs = (Cons x (Cons y ys)) /\ split xs = Split r xxs
    => genSort split join xs
         = join (Split r (map (genSort split join) xxs))  %(GenSortT1)%
. xs = (Cons x (Cons y Nil)) /\ split xs = Split r xxs
    => genSort split join xs
         = join (Split r (map (genSort split join) xxs))  %(GenSortT2)%
. xs = (Cons x Nil) \/ xs = Nil
    => genSort split join xs = xs                         %(GenSortF)%
. splitInsertionSort (Cons x xs)
    = Split x (Cons xs (Nil: List (List a)))              %(SplitInsertionSort)%
. joinInsertionSort (Split x (Cons xs (Nil: List (List a))))
    = insert x xs                                         %(JoinInsertionSort)%
. insertionSort xs
    = genSort splitInsertionSort joinInsertionSort xs     %(InsertionSort)%
. splitQuickSort (Cons x xs)
    = let (ys, zs) = partition (\t: ?a .! x < t) xs
      in Split x (Cons ys (Cons zs Nil))                  %(SplitQuickSort)%
. joinQuickSort (Split x (Cons ys (Cons zs Nil)))
    = ys ++ (Cons x zs)                                   %(JoinQuickSort)%
. quickSort xs = genSort splitQuickSort joinQuickSort xs  %(QuickSort)%
. splitSelectionSort xs = let x = minimum xs
  in Split x (Cons (delete x xs) (Nil: List(List a)))     %(SplitSelectionSort)%
. joinSelectionSort (Split x (Cons xs Nil)) = (Cons x xs) %(JoinSelectionSort)%
. selectionSort xs
    = genSort splitSelectionSort joinSelectionSort xs     %(SelectionSort)%
. def((length xs) div 2) /\ n = ((length xs) div 2)
    => splitMergeSort xs = let (ys,zs) = splitAt n xs
       in Split () (Cons ys (Cons zs Nil))                %(SplitMergeSort)%
. xs = (Nil: List a) => merge xs ys = ys                  %(MergeNil)%
. xs = (Cons v vs) /\ ys = (Nil: List a)
    => merge xs ys = xs                                   %(MergeConsNil)%
```

```
. xs = (Cons v vs) /\ ys = (Cons w ws) /\ (v < w)
     => merge xs ys = Cons v (merge vs ys)                 %(MergeConsConsT)%
. xs = (Cons v vs) /\ ys = (Cons w ws) /\ not (v < w)
     => merge xs ys = Cons w (merge xs ws)                 %(MergeConsConsF)%
. joinMergeSort (Split () (Cons ys (Cons zs Nil)))
     = merge ys zs                                         %(JoinMergeSort)%
. mergeSort xs = genSort splitMergeSort joinMergeSort xs   %(MergeSort)%
then
vars a: Ord;
     x,y: ?a;
     xs,ys: ?List a
preds __elem__ : ?a * ?List a;
      isOrdered: ?List a;
      permutation: ?List a * ?List a
. not x elem (Nil: List a)                                 %(ElemNil)%
. x elem (Cons y ys) <=> x = y \/ x elem ys                %(ElemCons)%
. isOrdered (Nil: List a)                                  %(IsOrderedNil)%
. isOrdered (Cons x (Nil: List a))                         %(IsOrderedCons)%
. isOrdered (Cons x (Cons y ys))
     <=> (x <= y)  /\ isOrdered(Cons y ys)           %(IsOrderedConsCons)%
. permutation ((Nil: List a), Nil)                         %(PermutationNil)%
. permutation (Cons x (Nil: List a), Cons y (Nil: List a))
     <=> x=y                                               %(PermutationCons)%
. permutation (Cons x xs, Cons y ys) <=>
     (x=y /\ permutation (xs, ys)) \/ (x elem ys
          /\ permutation(xs, Cons y (delete x ys)))        %(PermutationConsCons)%
then %implies
var a,b : Ord;
    xs, ys : ?List a;
. insertionSort xs = quickSort xs                          %(Theorem01)%
. insertionSort xs = mergeSort xs                          %(Theorem02)%
```

```
. insertionSort xs = selectionSort xs                    %(Theorem03)%
. quickSort xs = mergeSort xs                            %(Theorem04)%
. quickSort xs = selectionSort xs                        %(Theorem05)%
. mergeSort xs = selectionSort xs                        %(Theorem06)%
. isOrdered(insertionSort xs)                            %(Theorem07)%
. isOrdered(quickSort xs)                                %(Theorem08)%
. isOrdered(mergeSort xs)                                %(Theorem09)%
. isOrdered(selectionSort xs)                            %(Theorem10)%
. permutation(xs, insertionSort xs)                      %(Theorem11)%
. permutation(xs, quickSort xs)                          %(Theorem12)%
. permutation(xs, mergeSort xs)                          %(Theorem13)%
. permutation(xs, selectionSort xs)                      %(Theorem14)%
end
```

Apêndice D

Listagem das Provas Desenvolvidas em Isabelle/HOL

Este apêndice contém o código das provas para especificações com avaliação estrita desenvolvidas neste trabalho com o uso da linguagem HOL e verificadas com o provador de teorema Isabelle. Cada seção apresenta apenas os teoremas e o seus respectivos códigos de prova.

D.1 LazyPrelude_Bool.thy

```
theorem NotFalse1 : "ALL (x :: bool). Not' x = (~ x)"
by (auto)
ML "Header.record \"NotFalse1\""

theorem NotTrue1 : "ALL (x :: bool). ~ Not' x = x"
by (auto)
ML "Header.record \"NotTrue1\""

theorem notNot1 : "ALL (x :: bool). (~ x) = Not' x"
by (auto)
ML "Header.record \"notNot1\""

theorem notNot2 : "ALL (x :: bool). (~ ~ x) = (~ Not' x)"
apply(auto)
apply(case_tac x)
apply(auto)
done
ML "Header.record \"notNot2\""
end
```

D.2 LazyPrelude_Eq.thy

```
theorem DiffSymDef :
"ALL (x :: 'a partial). ALL (y :: 'a partial).
         x /= y = y /= x"
apply(auto)
apply(simp add: DiffDef)
apply(simp add: EqualSymDef)
apply(simp add: DiffDef)
apply(simp add: EqualSymDef)
done
ML "Header.record \"DiffSymDef\""

theorem DiffTDef :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x /= y = Not' (x ==' y)"
apply(auto)
apply(simp add: DiffDef)
apply(simp add: DiffDef)
done
ML "Header.record \"DiffTDef\""

theorem DiffFDef :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x /= y) = x ==' y"
apply(auto)
apply(simp add: DiffDef)
apply(simp add: NotFalse1)
apply(simp add: DiffDef)
done
ML "Header.record \"DiffFDef\""

theorem TE1 :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). ~ x ==' y --> ~ x = y"
by (auto)
ML "Header.record \"TE1\""

theorem TE2 :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). Not' (x ==' y) = (~ x ==' y)"
by (auto)
ML "Header.record \"TE2\""

theorem TE3 :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ Not' (x ==' y)) = x ==' y"
apply(auto)
apply(case_tac "x ==' y")
by (auto)
ML "Header.record \"TE3\""

theorem TE4 :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x ==' y) = (~ x ==' y)"
by (auto)
ML "Header.record \"TE4\""

theorem IUE1 : "makePartial () ==' makePartial ()"
by (auto)
ML "Header.record \"IUE1\""

theorem IUE2 : "~ makePartial () /= makePartial ()"
apply(simp add: DiffDef)
done
ML "Header.record \"IUE2\""

theorem IBE1 : "makePartial () ==' makePartial ()"
by (auto)
ML "Header.record \"IBE1\""

theorem IBE2 : "undefinedOp ==' undefinedOp"
by (auto)
ML "Header.record \"IBE2\""

theorem IBE4 : "~ makePartial () ==' undefinedOp"
apply (auto)
apply(simp add: EqualSymDef)
done
```

```
ML "Header.record \"IBE4\""

theorem IBE5 : "makePartial () /= undefinedOp"
apply(simp add: DiffDef)
apply(simp add: NotFalse1)
apply(simp add: EqualSymDef)
done
ML "Header.record \"IBE5\""

theorem IBE6 : "undefinedOp /= makePartial ()"
apply(simp add: DiffDef)
done
ML "Header.record \"IBE6\""

theorem IBE7 : "Not' (makePartial () ==' undefinedOp)"
apply(simp add: NotFalse1)
apply(simp add: EqualSymDef)
done
ML "Header.record \"IBE7\""

theorem IBE8 : "~ Not' Not' (makePartial () ==' undefinedOp)"
apply(simp add: EqualSymDef)
done
ML "Header.record \"IBE8\""
end
```

D.3 LazyPrelude_Ord.thy

```
theorem IOE01 : "makePartial LT ==' makePartial LT"
by (auto)
ML "Header.record \"IOE01\""

theorem IOE02 : "makePartial EQ ==' makePartial EQ"
by (auto)
ML "Header.record \"IOE02\""

theorem IOE03 : "makePartial GT ==' makePartial GT"
by (auto)
ML "Header.record \"IOE03\""

theorem IOE07 : "makePartial LT /= makePartial EQ"
apply(simp add: DiffDef)
done
ML "Header.record \"IOE07\""

theorem IOE08 : "makePartial LT /= makePartial GT"
apply(simp add: DiffDef)
done
ML "Header.record \"IOE08\""

theorem IOE09 : "makePartial EQ /= makePartial GT"
apply(simp add: DiffDef)
done
ML "Header.record \"IOE09\""

lemma LeIrreflContra : " x <' x ==> False"
by (auto)

theorem LeTAsymmetry :
"ALL (x :: 'a partial). ALL (y :: 'a partial).
         x <' y --> ~ y <' x"
apply(auto)
apply(rule ccontr)
apply(simp add: notNot2 NotTrue1)
apply(rule_tac x="x" in LeIrreflContra)
apply(rule_tac y = "y" in LeTTransitive)
by (auto)
ML "Header.record \"LeTAsymmetry\""

theorem GeIrreflexivity :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x ==' y --> ~ x >' y"
apply(auto)
```

```
apply(simp add: GeDef)
apply(simp add: EqualSymDef LeTAsymmetry)
done
ML "Header.record \"GeIrreflexivity\""

theorem GeTAsymmetry :
"ALL (x :: 'a partial). ALL (y :: 'a partial).
         x >' y --> ~ y >' x"
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
done
ML "Header.record \"GeTAsymmetry\""

theorem GeTTransitive :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial).
 ALL (z :: 'a partial). (x >' y) && (y >' z) --> x >' z"
apply(auto)
apply(simp add: GeDef)
apply(rule_tac x="z" and y="y" and z="x" in  LeTTransitive)
apply(auto)
apply(case_tac  "y <' x")
apply(auto)
apply(case_tac  "y <' x")
by(auto)
ML "Header.record \"GeTTransitive\""

theorem GeTTotal :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). ((x >' y) || (y >' x))
         || (x ==' y)"
apply(auto)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeFGeTEqTRel)
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
apply(simp add: EqualSymDef)
done
ML "Header.record \"GeTTotal\""

theorem LeqReflexivity : "ALL (x :: 'a partial). x <=' x"
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
done
ML "Header.record \"LeqReflexivity\""

lemma EqualL1 [rule_format]:
"ALL a b ab bb.
 (x ==' z) & ~ (x ==' z) \<longrightarrow> False"
by(auto)

lemma EqualL2 [rule_format]:
"ALL a b aa ab ba bb.
(x ==' y) & (y ==' z) \<longrightarrow> ~ (x ==' z)
         \<longrightarrow> False"
apply(simp add: EqualL1)
apply(auto)
apply(rule EqualTransT)
by(auto)

lemma EqualL3 [rule_format]:
"ALL a b aa ab ba bb.
~ (x ==' y) | ~ (y ==' z) | ~ (x ==' z)
         \<longrightarrow> False \<longrightarrow> False"
by(auto)

lemma Le1E [rule_format]:
"ALL a b aa ab ba bb.
(y ==' x) & (x <' z) \<longrightarrow> (y <' z)"
apply (auto)
apply(rule EqTOrdTSubstE)
by(auto)

lemma Le2 [rule_format]:
```

```
"ALL a b aa ab ba bb.
(x <' y) \<longrightarrow> ~ (x <' y)
        \<longrightarrow> False"
by auto

lemma Le3E [rule_format]:
"ALL a b aa ab ba bb.
(y ==' x) & (x <' z) \<longrightarrow> ~ (y <' z)
         \<longrightarrow> False"
apply (auto)
apply(rule EqTOrdTSubstE)
by(auto)

lemma Le3D [rule_format]:
"ALL a b aa ab ba bb.
(y ==' x) & (z <' x) \<longrightarrow> ~ (z <' y)
         \<longrightarrow> False"
apply (auto)
apply(rule EqTOrdTSubstD)
apply(auto)
done

lemma Le4E [rule_format]:
"ALL a b aa ab ba bb.
(y ==' x) & ~ (x <' z) \<longrightarrow> ~ (y <' z)"
apply (auto)
apply(rule Le3E)
apply(auto)
apply(simp add: EqualSymDef)
done

lemma Le4D [rule_format]:
"ALL a b aa ab ba bb.
(y ==' x) & ~ (z <' x) \<longrightarrow> ~ (z <' y)"
apply (auto)
apply(rule Le3D)
apply(auto)
apply(simp add: EqualSymDef)
done

lemma Le5 [rule_format]:
"ALL a b aa ab ba bb.
~ (x <' y) \<longrightarrow> (x <' y)
         \<longrightarrow> False"
by auto

lemma Le6E [rule_format]:
"ALL a b aa ab ba bb.
(y ==' x) & ~ (x <' z) \<longrightarrow> (y <' z)
         \<longrightarrow> False"
apply (auto)
apply(rule Le5)
apply(rule EqTOrdFSubstE)
apply(auto)
done

lemma Le7 [rule_format]:
"ALL a b aa ab ba bb.
x <' y & ~ x <' y \<longrightarrow> False"
by auto

lemma Le8 [rule_format]:
"ALL a b aa ab ba bb.
z ==' y & x <' y \<longrightarrow> x <' z"
apply auto
apply(rule EqTOrdTSubstD)
apply(rule conjI)
by(auto)

lemma Le9 [rule_format]:
"ALL a b aa ab ba bb.
x <' y & y ==' z \<longrightarrow> ~ x <' z
         \<longrightarrow> False"
apply auto
apply(rule Le8)
apply(auto)
apply (simp add: EqualSymDef)
done

lemma Le10 [rule_format]:
"ALL a b aa ab ba bb.
y <' z & x ==' y \<longrightarrow> ~ x <' z
         \<longrightarrow> False"
apply auto
```

```
apply(rule EqTOrdTSubstE)
apply(rule conjI)
by(auto)

lemma Le11 [rule_format]:
"ALL a b aa ab ba bb.
z <' y & y <' x \<longrightarrow> ~ z <' x
         \<longrightarrow> False"
apply(auto)
apply(rule LeTTransitive)
apply(auto)
done

lemma Le12 [rule_format]:
"ALL a b aa ab ba bb.
y <' x & y ==' z \<longrightarrow> ~ z <' x
         \<longrightarrow> False"
apply(auto)
apply(rule EqTOrdTSubstE)
apply(rule conjI)
apply(auto)
apply(simp add: EqualSymDef)
done

lemma Le13 [rule_format]:
"ALL a b aa ab ba bb.
x <' z & z <' y \<longrightarrow> ~ x <' y
         \<longrightarrow> False"
apply(auto)
apply(rule LeTTransitive)
apply(rule conjI)
apply(auto)
done

lemma Le14 [rule_format]:
"ALL a b aa ab ba bb.
~ x <' x"
by(auto)

lemma Le15 [rule_format]:
"ALL a b aa ab ba bb.
x <' z & z <' y \<longrightarrow> x <' y & y <' x
         \<longrightarrow> False"
apply(auto)
apply(simp add: LeTAsymmetry)
done

lemma Le16 [rule_format]:
"ALL a b aa ab ba bb.
x ==' y & z <' y \<longrightarrow> z <' x & ~ z <' x
         \<longrightarrow> False"
by(auto)

lemma Le17 [rule_format]:
"ALL a b aa ab ba bb.
z <' y & x ==' y \<longrightarrow> z <' x"
apply(auto)
apply(rule EqTOrdTSubstD)
apply(rule conjI)
apply(auto)
done

lemma Le18 [rule_format]:
"ALL a b aa ab ba bb.
x <' z & ~ x <' z \<longrightarrow> False"
by(auto)

lemma Le19 [rule_format]:
"ALL a b aa ab ba bb.
x ==' y & y <' z \<longrightarrow> ~ x <' z
         \<longrightarrow> False"
apply(auto)
apply(rule EqTOrdTSubstE)
apply(auto)
done

lemma Le20 [rule_format]:
"ALL a b aa ab ba bb.
x ==' y & z <' y \<longrightarrow> ~ z <' x
         \<longrightarrow> False"
apply(auto)
apply(rule EqTOrdTSubstD)
apply(auto)
done
```

```
theorem LeqTTransitive :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial).
 ALL (z :: 'a partial). (x <=' y) && (y <=' z)
         --> x <=' z"
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "y <' z")
apply(auto)
apply(case_tac "x <' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
(*Here we needed the first aux lemma*)
apply(rule Le18)
apply(rule conjI)
apply(rule LeTTransitive)
apply(auto)
apply(case_tac "y ==' z")
apply(auto)
apply(case_tac "x <' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(rule Le9)
apply(rule conjI)
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y <' z")
apply(auto)
apply(case_tac "x <' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(rule Le19)
apply(auto)
apply(case_tac "y ==' z")
apply(auto)
apply(case_tac "x <' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(rule EqualL2)
by(auto)
ML "Header.record \"LeqTTransitive\""

theorem LeqTTotal :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (x <=' y) && (y <=' x)
         = x ==' y"
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
apply(simp add: LeTAsymmetry)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: LeqDef)
apply(simp add: OrDef)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef EqualL1)
done
ML "Header.record \"LeqTTotal\""

theorem GeqReflexivity : "ALL (x :: 'a partial).
         x >=' x"
apply(auto)
apply(simp add: GeqDef)
apply(simp add: OrDef)
apply(simp add: AndSym)
```

```
done
ML "Header.record \"GeqReflexivity\""

theorem GeqTTransitive :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial).
 ALL (z :: 'a partial). (x >=' y) && (y >=' z)
         --> x >=' z"
apply(auto)
apply(simp add: GeqDef)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "y >' z")
apply(auto)
apply(case_tac "x >' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
(*Here we needed the first aux lemma*)
apply(simp add: GeDef)
apply(rule Le18)
apply(rule conjI)
apply(rule LeTTransitive)
apply(auto)
apply(case_tac "y ==' z")
apply(auto)
apply(case_tac "x >' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(simp add: GeDef)
apply(rule Le10)
apply(rule conjI)
apply(simp add: EqualSymDef)+
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y >' z")
apply(auto)
apply(case_tac "x >' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(simp add: GeDef)
apply(rule Le20)
apply(rule conjI)
apply(simp add: EqualSymDef)+
apply(case_tac "y ==' z")
apply(auto)
apply(case_tac "x >' z")
apply(auto)
apply(case_tac "x ==' z")
apply(auto)
apply(rule EqualL2)
by(auto)
ML "Header.record \"GeqTTransitive\""

theorem GeqTTotal :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (x >=' y) && (y >=' x)
        = x ==' y"
apply(auto)
apply(simp add: GeqDef)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeqDef)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef EqualL1)
```

```
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef EqualL1)
done
ML "Header.record \"GeqTTotal\""

theorem LeTGeTRel :
"ALL (x :: 'a partial). ALL (y :: 'a partial).
         x <' y = y >' x"
apply(auto)
apply(simp add: GeDef)+
done
ML "Header.record \"LeTGeTRel\""

theorem LeFGeFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x <' y) = (~ y >' x)"
apply(auto)
apply(simp add: GeDef)+
done
ML "Header.record \"LeFGeFRel\""

theorem LeqTGetTRel :
"ALL (x :: 'a partial). ALL (y :: 'a partial).
         x <=' y = y >=' x"
apply(auto)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: GeDef)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
done
ML "Header.record \"LeqTGetTRel\""

theorem LeqFGetFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x <=' y) = (~ y >=' x)"
apply(auto)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: GeDef)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
```

```
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "y >' x")
apply(auto)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
done
ML "Header.record \"LeqFGetFRel\""

theorem GeTLeTRel :
"ALL (x :: 'a partial). ALL (y :: 'a partial).
         x >' y = y <' x"
apply(auto)
apply(simp add: GeDef)+
done
ML "Header.record \"GeTLeTRel\""

theorem GeFLeFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x >' y) = (~ y <' x)"
apply(auto)
apply(simp add: GeDef)+
done
ML "Header.record \"GeFLeFRel\""

theorem GeqTLeqTRel :
"ALL (x :: 'a partial). ALL (y :: 'a partial).
         x >=' y = y <=' x"
apply(auto)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: GeDef)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
done
ML "Header.record \"GeqTLeqTRel\""

theorem GeqFLeqFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x >=' y) = (~ y <=' x)"
apply(auto)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
```

```
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: GeDef)
apply(case_tac "x ==' y")
apply(auto)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "y ==' x")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "y <' x")
apply(auto)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(case_tac "y ==' x")
apply(auto)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
done
ML "Header.record \"GeqFLeqFRel\""

theorem LeqTGeFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x <=' y = (~ x >' y)"
apply(auto)
apply(simp add: GeqDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: EqualSymDef)
apply(simp add: GeDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: LeFGeTEqTRel)
apply(simp add: EqFSOrdRel)
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeTAsymmetry)
apply(simp add: GeDef)
done
ML "Header.record \"LeqTGeFRel\""

theorem LeqFGeTRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x <=' y) = x >' y"
apply(auto)
apply(simp add: GeDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: LeFGeTEqTRel)
apply(simp add: EqFSOrdRel)
apply(simp add: GeDef)
apply(simp add: GeDef LeqDef)
apply(simp add: OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(simp add: LeTAsymmetry)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef LeTAsymmetry)
done
ML "Header.record \"LeqFGeTRel\""

theorem GeTLeFEqFRel :
"ALL (x :: 'a partial).
```

```
 ALL (y :: 'a partial). x >' y = (~ x <' y & ~ x ==' y)"
apply(auto)
apply(simp add: GeDef LeTAsymmetry)
apply(simp add: GeDef)
apply(simp add: EqualSymDef LeTAsymmetry)
apply(simp add: EqFSOrdRel)
done
ML "Header.record \"GeTLeFEqFRel\""

theorem GeFLeTEqTRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x >' y) = (x <' y | x ==' y)"
apply(auto)
apply(simp add: LeTGeFEqFRel)
apply(simp add: GeDef LeTAsymmetry)
apply(simp add: GeDef)
apply(simp add: EqualSymDef)
done
ML "Header.record \"GeFLeTEqTRel\""

theorem GeqTLeFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x >=' y = (~ x <' y)"
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(simp add: GeDef LeTAsymmetry)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(simp add: GeDef LeTAsymmetry)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: LeFGeTEqTRel)
apply(simp add: EqFSOrdRel)
apply(simp add: GeDef LeTAsymmetry)
done
ML "Header.record \"GeqTLeFRel\""

theorem GeqFLeTRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x >=' y) = x <' y"
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(simp add: GeDef LeTAsymmetry)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: LeFGeTEqTRel)
apply(simp add: EqFSOrdRel)
apply(simp add: GeDef LeTAsymmetry)
apply(rule disjE)
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(simp add: GeDef LeTAsymmetry)
apply(case_tac "x ==' y")
by(auto)
ML "Header.record \"GeqFLeTRel\""

theorem LeqTLeTEqTRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x <=' y = (x <' y | x ==' y)"
apply(auto)
apply(simp add: LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(simp add: LeqDef OrDef LeTAsymmetry)+
done
ML "Header.record \"LeqTLeTEqTRel\""

theorem LeqFLeFEqFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x <=' y)
        = (~ x <' y & ~ x ==' y)"
apply(auto)
apply(simp add: LeqDef OrDef)+
done
ML "Header.record \"LeqFLeFEqFRel\""
```

```
theorem GeqTGeTEqTRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x >=' y = (x >' y | x ==' y)"
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(simp add: GeqDef OrDef)+
apply(case_tac "x >' y")
apply(auto)
done
ML "Header.record \"GeqTGeTEqTRel\""

theorem GeqFGeFEqFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (~ x >=' y)
       = (~ x >' y & ~ x ==' y)"
apply(auto)
apply(simp add: GeqDef OrDef)+
apply(case_tac "x >' y")
apply(auto)
apply(simp add: GeqDef OrDef)+
done
ML "Header.record \"GeqFGeFEqFRel\""

theorem LeTGeqFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x <' y = (~ x >=' y)"
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(simp add: GeDef LeTAsymmetry)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeqDef OrDef)
apply(case_tac "x >' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeFGeTEqTRel)
apply(simp add: EqFSOrdRel)
apply(simp add: GeDef LeTAsymmetry)
apply(rule disjE)
by(auto)
ML "Header.record \"LeTGeqFRel\""

theorem GeTLeqFRel :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x >' y = (~ x <=' y)"
apply(auto)
apply(simp add: GeDef LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(simp add: LeTAsymmetry)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: EqualSymDef)
apply(simp add: LeqDef OrDef)
apply(case_tac "x <' y")
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: GeDef)
apply(simp add: LeFGeTEqTRel)
apply(simp add: EqFSOrdRel)
apply(simp add: GeDef LeTAsymmetry)
done
ML "Header.record \"GeTLeqFRel\""

theorem LeLeqDiff :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x <' y
       = (x <=' y) && (x /= y)"
apply(auto)
apply(simp add: LeqDef OrDef)
apply(simp add: DiffDef)
apply(case_tac "x ==' y")
apply(auto)
apply(simp add: LeqDef OrDef)
apply(simp add: DiffDef)
apply(case_tac "x <' y")
```

```
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
done
ML "Header.record \"LeLeqDiff\""

theorem MaxSym :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). X_max x y ==' y
         = X_max y x ==' y"
by (auto)
ML "Header.record \"MaxSym\""

theorem MinSym :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). X_min x y ==' y
        = X_min y x ==' y"
by (auto)
ML "Header.record \"MinSym\""

theorem TO1 :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). (x ==' y | x <' y) = x <=' y"
apply(auto)
apply(simp add: LeqDef OrDef)+
apply(case_tac "x ==' y")
apply(auto)
done
ML "Header.record \"TO1\""

theorem TO2 :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x ==' y --> ~ x <' y"
by (auto)
ML "Header.record \"TO2\""

theorem TO3 :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). Not' Not' (x <' y)
         | Not' (x <' y)"
by (auto)
ML "Header.record \"TO3\""

theorem TO4 :
"ALL (x :: 'a partial).
 ALL (y :: 'a partial). x <' y --> Not' (x ==' y)"
apply(auto)
apply(case_tac "x ==' y")
apply(auto)
done
ML "Header.record \"TO4\""

theorem TO5 :
"ALL (w :: 'a partial).
 ALL (x :: 'a partial).
 ALL (y :: 'a partial).
 ALL (z :: 'a partial). (x <' y & y <' z) & z <' w
         --> x <' w"
apply(auto)
apply(rule_tac y="z" in LeTTransitive)
apply(auto)
apply(rule_tac y="y" in LeTTransitive)
by auto
ML "Header.record \"TO5\""

theorem TO6 :
"ALL (x :: 'a partial).
 ALL (z :: 'a partial). z <' x --> Not' (x <' z)"
apply(auto)
apply(case_tac "x <' z")
apply(auto)
apply(simp add: LeTAsymmetry)
done
ML "Header.record \"TO6\""

theorem TO7 :
"ALL (x :: 'a partial). ALL (y :: 'a partial).
         x <' y = y >' x"
apply(auto)
apply(simp add: GeDef)+
done
ML "Header.record \"TO7\""

theorem IUO01 : "makePartial () <=' makePartial ()"
```

```
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IUO01\""

theorem IUO02 : "~ makePartial () <' makePartial ()"
by (auto)
ML "Header.record \"IUO02\""

theorem IUO03 : "makePartial () >=' makePartial ()"
apply(simp add: GeqDef OrDef)
apply(case_tac "makePartial () >' makePartial ()")
apply(auto)
done
ML "Header.record \"IUO03\""

theorem IUO04 : "~ makePartial () >' makePartial ()"
apply(simp add: GeDef)
done
ML "Header.record \"IUO04\""

theorem IUO05 :
"X_max (makePartial ()) (makePartial ())
        ==' makePartial ()"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IUO05\""

theorem IUO06 :
"X_min (makePartial ()) (makePartial ())
         ==' makePartial ()"
apply(simp add: MinXDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IUO06\""

theorem IUO07 :
"compare (makePartial ()) (makePartial ())
         ==' makePartial EQ"
by (auto)
ML "Header.record \"IUO07\""

theorem IOO16 : "makePartial LT <=' makePartial EQ"
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IOO16\""

theorem IOO17 : "makePartial EQ <=' makePartial GT"
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IOO17\""

theorem IOO18 : "makePartial LT <=' makePartial GT"
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IOO18\""

theorem IOO19 : "makePartial EQ >=' makePartial LT"
apply(simp add: GeqDef OrDef GeDef)
done
ML "Header.record \"IOO19\""

theorem IOO20 : "makePartial GT >=' makePartial EQ"
apply(simp add: GeqDef OrDef GeDef)
done
ML "Header.record \"IOO20\""

theorem IOO21 : "makePartial GT >=' makePartial LT"
apply(simp add: GeqDef OrDef GeDef)
done
ML "Header.record \"IOO21\""

theorem IOO22 : "makePartial EQ >' makePartial LT"
apply(simp add: GeDef OrDef)
done
ML "Header.record \"IOO22\""

theorem IOO23 : "makePartial GT >' makePartial EQ"
apply(simp add: GeDef OrDef)
done
ML "Header.record \"IOO23\""

theorem IOO24 : "makePartial GT >' makePartial LT"
apply(simp add: GeDef OrDef)
```

```
done
ML "Header.record \"IO024\""

theorem IO025 :
"X_max (makePartial LT) (makePartial EQ)
        ==' makePartial EQ"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO025\""

theorem IO026 :
"X_max (makePartial EQ) (makePartial GT)
         ==' makePartial GT"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO026\""

theorem IO027 :
"X_max (makePartial LT) (makePartial GT)
         ==' makePartial GT"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO027\""

theorem IO028 :
"X_min (makePartial LT) (makePartial EQ)
         ==' makePartial LT"
apply(simp add: MinXDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO028\""

theorem IO029 :
"X_min (makePartial EQ) (makePartial GT)
         ==' makePartial EQ"
apply(simp add: MinXDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO029\""

theorem IO030 :
"X_min (makePartial LT) (makePartial GT)
        ==' makePartial LT"
apply(simp add: MinXDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IO030\""

theorem IO031 :
"compare (makePartial LT) (makePartial LT)
        ==' makePartial EQ"
by (auto)
ML "Header.record \"IO031\""

theorem IO032 :
"compare (makePartial EQ) (makePartial EQ)
        ==' makePartial EQ"
by (auto)
ML "Header.record \"IO032\""

theorem IO033 :
"compare (makePartial GT) (makePartial GT)
         ==' makePartial EQ"
by (auto)
ML "Header.record \"IO033\""

theorem IB06 : "~ undefinedOp >=' makePartial ()"
apply(simp add: GeqDef OrDef GeDef)
apply (case_tac "makePartial () <' undefinedOp")
apply(auto)
apply(simp add: LeTGeFEqFRel)
apply(simp add: GeDef)
done
ML "Header.record \"IB06\""

theorem IB07 : "makePartial () >=' undefinedOp"
apply(simp add: GeqDef OrDef GeDef)
done
ML "Header.record \"IB07\""

theorem IB08 : "~ makePartial () <' undefinedOp"
```

```
apply(simp add: LeFGeTEqTRel)
apply(simp add: GeDef)
done
ML "Header.record \"IBO8\""

theorem IBO9 :
"X_max undefinedOp (makePartial ()) ==' makePartial ()"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IBO9\""

theorem IBO10 :
"X_min undefinedOp (makePartial ()) ==' undefinedOp"
apply(simp add: MaxYDef)
apply(simp add: LeqDef OrDef)
done
ML "Header.record \"IBO10\""

theorem IBO11 :
"compare (makePartial ()) (makePartial ()) ==' makePartial EQ"
by (auto)
ML "Header.record \"IBO11\""

theorem IBO12 :
"compare undefinedOp undefinedOp ==' makePartial EQ"
by (auto)
ML "Header.record \"IBO12\""
end
```

D.4 LazyPrelude_Maybe.thy

```
theorem IME02 : "makePartial Nothing
        ==' makePartial Nothing"
by (auto)
ML "Header.record \"IME02\""

theorem IMO03 :
"ALL (x :: 'o partial). ~ makePartial Nothing >=' Just(x)"
apply(rule allI)
apply(simp only: GeqDef)
apply(simp only: GeDef OrDef)
apply(case_tac "Just(x) <' makePartial Nothing")
apply(auto)
done
ML "Header.record \"IMO03\""

theorem IMO04 :
"ALL (x :: 'o partial). Just(x) >=' makePartial Nothing"
apply(rule allI)
apply(simp only: GeqDef)
apply(simp only: GeDef OrDef)
apply(case_tac "makePartial Nothing <' Just(x)")
apply(auto)
done
ML "Header.record \"IMO04\""

theorem IMO05 :
"ALL (x :: 'o partial). ~ Just(x) <' makePartial Nothing"
apply(rule allI)
apply(case_tac "Just(x) <' makePartial Nothing")
apply(auto)
done
ML "Header.record \"IMO05\""

theorem IMO06 :
"ALL (x :: 'o partial).
 compare (makePartial Nothing) (Just(x)) ==' makePartial EQ
 = makePartial Nothing ==' Just(x)"
by (auto)
ML "Header.record \"IMO06\""

theorem IMO07 :
```

```
"ALL (x :: 'o partial).
 compare (makePartial Nothing) (Just(x)) ==' makePartial LT
 = makePartial Nothing <' Just(x)"
by (auto)
ML "Header.record \"IMO07\""

theorem IMO08 :
"ALL (x :: 'o partial).
 compare (makePartial Nothing) (Just(x)) ==' makePartial GT
 = makePartial Nothing >' Just(x)"
apply(rule allI)+
apply(simp add: GeDef)
done
ML "Header.record \"IMO08\""

theorem IMO09 :
"ALL (x :: 'o partial).
 makePartial Nothing <=' Just(x) =
 X_max (makePartial Nothing) (Just(x)) ==' Just(x)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IMO09\""

theorem IMO10 :
"ALL (x :: 'o partial).
 Just(x) <=' makePartial Nothing =
 X_max (makePartial Nothing) (Just(x))
        ==' makePartial Nothing"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IMO10\""

theorem IMO11 :
"ALL (x :: 'o partial).
 makePartial Nothing <=' Just(x) =
 X_min (makePartial Nothing) (Just(x)) ==' makePartial Nothing"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IMO11\""

theorem IMO12 :
"ALL (x :: 'o partial).
 Just(x) <=' makePartial Nothing =
 X_min (makePartial Nothing) (Just(x)) ==' Just(x)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IMO12\""
end
```

D.5 LazyPrelude_Either.thy

```
theorem IE004 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial). ~ Left'(x) >=' Right'(z)"
apply(rule allI)
apply(simp only: GeqDef)
apply(simp only: GeDef OrDef)
apply(case_tac "Right'(y) <' Left'(x)")
apply(auto)
done
ML "Header.record \"IE004\""

theorem IE005 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial). Right'(z) >=' Left'(x)"
apply(rule allI)
apply(simp only: GeqDef)
apply(simp only: GeDef OrDef)
apply(case_tac "Left'(x) <' Right'(y)")
apply(auto)
```

```
done
ML "Header.record \"IE005\""

theorem IE006 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial). ~ Right'(z) <' Left'(x)"
apply(rule allI)
apply(case_tac "Right'(y) <' Left'(x)")
apply(auto)
done
ML "Header.record \"IE006\""

theorem IE007 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial).
 compare (Left'(x)) (Right'(z)) ==' makePartial EQ =
 Left'(x) ==' Right'(z)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE007\""

theorem IE008 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial).
 compare (Left'(x)) (Right'(z)) ==' makePartial LT =
 Left'(x) <' Right'(z)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE008\""

theorem IE009 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial).
 compare (Left'(x)) (Right'(z)) ==' makePartial GT =
 Left'(x) >' Right'(z)"
apply(rule allI)+
apply(simp add: GeDef)
done
ML "Header.record \"IE009\""

theorem IE010 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial).
 Left'(x) <=' Right'(z) =
 X_max (Left'(x)) (Right'(z)) ==' Right'(z)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE010\""

theorem IE011 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial).
 Right'(z) <=' Left'(x) = X_max (Left'(x)) (Right'(z))
        ==' Left'(x)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE011\""

theorem IE012 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial).
 Left'(x) <=' Right'(z) = X_min (Left'(x)) (Right'(z))
        ==' Left'(x)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE012\""

theorem IE013 :
"ALL (x :: 'o partial).
 ALL (z :: 'oo partial).
 Right'(z) <=' Left'(x) =
 X_min (Left'(x)) (Right'(z)) ==' Right'(z)"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IE013\""
end
```

D.6 LazyPrelude_ListNoNumbers.thy

```
theorem PartitionProp :
"ALL p.
 ALL xs.
 partition p xs =
 (X_filter p xs, X_filter (X__o__X (Not__X, p)) xs)"
apply(auto)
apply(simp only: Partition)
apply(induct_tac xs)
apply(case_tac "p a")
apply(simp only: FoldrCons)
apply(simp only: FilterConsF)
apply(auto)
apply(simp add: FilterConsT)
apply(simp add: FoldrCons)
apply(simp only: FilterConsT)
done
ML "Header.record \"PartitionProp\""
end
```

D.7 LazyPrelude_ListNoNumbers_ E4.thy

```
theorem ILE01 : "Nil' ==' Nil' = True'"
by (auto)
ML "Header.record \"ILE01\""

theorem IL001 : "Nil' <' Nil' = False'"
by (auto)
ML "Header.record \"IL001\""

theorem IL002 : "Nil' <=' Nil' = True'"
by (auto)
ML "Header.record \"IL002\""

theorem IL003 : "Nil' >' Nil' = False'"
by (auto)
ML "Header.record \"IL003\""

theorem IL004 : "Nil' >=' Nil' = True'"
by (auto)
ML "Header.record \"IL004\""

theorem IL008 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons z zs <=' X_Cons w ws =
 (X_Cons z zs <' X_Cons w ws)
 || (X_Cons z zs ==' X_Cons w ws)"
apply(rule allI)+
apply(simp only: LeqDef)
done
ML "Header.record \"IL008\""

theorem IL009 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs. X_Cons z zs >' X_Cons w ws
 = X_Cons w ws <' X_Cons z zs"
apply(rule allI)+
apply(case_tac "X_Cons z zs >' X_Cons w ws")
apply(simp only: GeFLeFRel)
apply(simp only: GeTLeTRel)
done
ML "Header.record \"IL009\""
```

```
theorem ILO10 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons z zs >=' X_Cons w ws =
 (X_Cons z zs >' X_Cons w ws)
 || (X_Cons z zs ==' X_Cons w ws)"
apply(rule allI)+
apply(simp only: GeqDef)
done
ML "Header.record \"ILO10\""

theorem ILO11 : "compare Nil' Nil' ==' EQ
 = Nil' ==' Nil'"
by (auto)
ML "Header.record \"ILO11\""

theorem ILO12 : "compare Nil' Nil' ==' LT
 = Nil' <' Nil'"
by (auto)
ML "Header.record \"ILO12\""

theorem ILO13 : "compare Nil' Nil' ==' GT
 = Nil' >' Nil'"
by (auto)
ML "Header.record \"ILO13\""

theorem ILO14 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 compare (X_Cons z zs) (X_Cons w ws) ==' EQ =
 X_Cons z zs ==' X_Cons w ws"
apply(rule allI)+
apply(simp only: CmpEQDef)
done
ML "Header.record \"ILO14\""

theorem ILO15 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 compare (X_Cons z zs) (X_Cons w ws) ==' LT =
 X_Cons z zs <' X_Cons w ws"
apply(rule allI)+
apply(simp only: CmpLTDef)
done
ML "Header.record \"ILO15\""

theorem ILO16 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 compare (X_Cons z zs) (X_Cons w ws) ==' GT =
 X_Cons z zs >' X_Cons w ws"
apply(rule allI)+
apply(simp only: CmpGTDef)
done
ML "Header.record \"ILO16\""

theorem ILO17 : "X_max Nil' Nil' ==' Nil' = Nil' <=' Nil'"
by (auto)
ML "Header.record \"ILO17\""

theorem ILO18 : "X_min Nil' Nil' ==' Nil' = Nil' <=' Nil'"
by (auto)
ML "Header.record \"ILO18\""

theorem ILO19 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons z zs <=' X_Cons w ws =
 X_max (X_Cons z zs) (X_Cons w ws) ==' X_Cons w ws"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"ILO19\""
```

```
theorem IL020 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons w ws <=' X_Cons z zs =
 X_max (X_Cons z zs) (X_Cons w ws) ==' X_Cons z zs"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IL020\""

theorem IL021 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons z zs <=' X_Cons w ws =
 X_min (X_Cons z zs) (X_Cons w ws) ==' X_Cons z zs"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IL021\""

theorem IL022 :
"ALL w.
 ALL ws.
 ALL z.
 ALL zs.
 X_Cons w ws <=' X_Cons z zs =
 X_min (X_Cons z zs) (X_Cons w ws) ==' X_Cons w ws"
apply(rule allI)+
apply(simp add: LeqDef)
done
ML "Header.record \"IL022\""

theorem FoldlDecomp :
"ALL e.
 ALL i.
 ALL ts.
 ALL ys. X_foldl i e (ys ++' ts)
 = X_foldl i (X_foldl i e ys) ts"
oops
ML "Header.record \"FoldlDecomp\""

theorem MapDecomp :
"ALL f.
 ALL xs. ALL zs. X_map f (xs ++' zs)
 = X_map f xs ++' X_map f zs"
apply(auto)
apply(induct_tac xs)
apply(auto)
apply(simp add: MapCons XPlusXPlusCons)
done
ML "Header.record \"MapDecomp\""

theorem MapFunctor :
"ALL f.
 ALL g. ALL xs. X_map (X__o__X (g, f)) xs
 = X_map g (X_map f xs)"
apply(auto)
apply(induct_tac xs)
apply(auto)
apply(simp add: MapNil MapCons Comp1)
done
ML "Header.record \"MapFunctor\""

theorem FilterProm :
"ALL f.
 ALL p.
 ALL xs.
 X_filter p (X_map f xs)
 = X_map f (X_filter (X__o__X (p, f)) xs)"
apply(auto)
apply(induct_tac xs)
apply(auto)
apply(case_tac "p(f a)")
apply(auto)
apply(simp add: MapCons)
apply(simp add: FilterConsT)
apply(simp add: MapCons)
apply(simp add: FilterConsT)
done
ML "Header.record \"FilterProm\""
end
```

D.8 LazyPrelude_Char.thy

```
theorem ICE02 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 Not'
 (restrictOp (makePartial (ord'(makeTotal x))) (defOp x) =='
  restrictOp (makePartial (ord'(makeTotal y))) (defOp y)) =
 x /= y"
apply(auto)
apply(simp add: DiffDef)+
done
ML "Header.record \"ICE02\""

theorem ICO05 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 ((defOp x & defOp y) &
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal x)) <=_3
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal y))) =
 x <=_4 y"
oops
ML "Header.record \"ICO05\""

theorem ICO06 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 ((defOp x & defOp y) &
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal x)) >_3
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal y))) =
 x >_4 y"
oops
ML "Header.record \"ICO06\""

theorem ICO07 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 ((defOp x & defOp y) &
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal x)) >=_3
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal y))) =
 x >=_4 y"
oops
ML "Header.record \"ICO07\""

theorem ICO01 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 compare x y ==' makePartial EQ =
 restrictOp (makePartial (ord'(makeTotal x))) (defOp x) =='
 restrictOp (makePartial (ord'(makeTotal y))) (defOp y)"
by auto

ML "Header.record \"ICO01\""

theorem ICO02 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 compare x y ==' makePartial LT =
 ((defOp x & defOp y) &
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal x)) <_3
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal y)))"
oops
ML "Header.record \"ICO02\""

theorem ICO03 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 compare x y ==' makePartial GT =
 ((defOp x & defOp y) &
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal x)) >_3
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal y)))"
oops
ML "Header.record \"ICO03\""

theorem ICO08 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 ((defOp x & defOp y) &
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal x)) <=_3
```

```
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal y))) =
 X_maxX4 x y ==' y"
oops
ML "Header.record \"ICO08\""

theorem ICO09 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 ((defOp y & defOp x) &
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal y)) <=_3
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal x))) =
 X_maxX4 x y ==' x"
oops
ML "Header.record \"ICO09\""

theorem ICO10 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 ((defOp x & defOp y) &
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal x)) <=_3
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal y))) =
 X_minX4 x y ==' x"
oops
ML "Header.record \"ICO10\""

theorem ICO11 :
"ALL (x :: Char partial).
 ALL (y :: Char partial).
 ((defOp y & defOp x) &
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal y)) <=_3
  (X_gn_inj :: X_Nat => Rat) (ord'(makeTotal x))) =
 X_minX4 x y ==' y"
oops
ML "Header.record \"ICO11\""
end
```