STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor SOČ: 18 informatika

DKIDE

Vývojové prostředí pro kernelový vývoj

DKIDE

Integrated development enviroment for kernel development

Autor: Jan Tomšů

Škola: Střední průmyslová škola elektrotechnická, Praha 10, V Úžlabině 320

Kraj: Praha

Praha 2015

Prohlášení

Prohlašuji, že jsem svou práci SOČ vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v seznamu vloženém v práci SOČ. Prohlašuji, že tištěná verze a elektronická verze soutěžní práce SOČ jsou shodné. Nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) v platném znění.

V ……. dne …… podpis: ……

Poděkování

Děkuji Ing. Miroslavu Škopovi za rady a připomínky poskytované během psaní této práce.

Anotace

Projekt DKIDE vznikl s cílem vytvořit IDE uzpůsobené pro kernelový vývoj a zaplnit tak mezeru na open source scéně věnující se kernelovému vývoji. Dalším cílem bylo vytvořit experimentální ovládací prostředí pomocí REPLu interpretovaného jazyka Rose. V dnešní době je to jediné existující IDE plně ovládané z REPLu na světě.

Klíčová slova: IDE, REPL, Interpret, Kernel, OS

Annotation

DKIDE project was designed to create IDE for kernel development and filling the gap in the open source scene dedicated to kernel development. Another goal is to create experimental user interface via Rose language REPL. At this moment, there isn't any another IDE with this approach in the world.

Keywords: IDE, REPL, Interpreter, Kernel, OS

1 Úvod

Projekt na tvorbu IDE podporující kernelový vývoj vznikl poté, co jsem se začal aktivně vzdělávat v oboru tvorby OS a narazil jsem na problém absence IDE, které by bylo určeno pro tento účel. Samozřejmě stávající IDE je možné nastavit tak, že je lze použít na kernelový vývoj (příkladem budiž KDevelop, Code::Block či Visual Studio), žádné ale tuto možnost nenabízí tak říkajíc od spuštění. K dnešnímu dni už minulé tvrzení není pravda, protože bylo vydáno LinK IDE určené pro vývoj linuxového kernelu (LinK IDE je plugin pro Eclipse IDE). V době, kdy jsem začal přemýšlet o tvorbě vlastního IDE ještě LinK IDE neexistovalo. Oproti LinK IDE se DKIDE pokouší o experiment s ovládáním. Místo klasického interaktivního rozhraní obsahuje pouze vizualizační a editační prvky. Ovládání a nastavování celého prostředí se přesunulo do REPLu mnou vytvořeného minimalistického funkcionálního jazyka Rose s podporou paralelismu bez nutnosti zamykat data díky persistentním datům. Dalším vylepšením je také zbavení uživatele nutnosti ukládat projekt. DKIDE automaticky synchronizuje veškeré změny s projektem uloženém na HDD, takže nehrozí nikdy ztráta dat.
Poznámka: Zdrojové kódy mého projektu byly psány na Linuxu, takže používají unixový standard řádkových zlomů. Při pokusu si zdrojové kódy přečíst v editoru, který je nepodporuje , uvidíme text bez řádkových zlomů.

2 Použité technologie při výrobě aplikace

Při tvorbě DKIDE bylo použito čistě funkcionálního jazyka Haskell. Volba padla na Haskell ze dvou důvodů. První důvod byla elegantnost funkcionálního kódu, pohodlnost a rychlost vývoje daná vyšší úrovní abstrakce funkcionálního paradigmatu. Aktuální velikost mého projektu, co se týče počtu řádek, tvoří pouhých 3 500 řádků kódu. Jedná se o skutečně nízké číslo vzhledem k tomu, že je v projektu obsažen i interpret jazyka. Pokud by bylo použito libovolného imperativního jazyka, délka kódu by byla zhruba sedmínásobná až desetinásobná. Tento efekt je opět způsoben vysokou úrovní abstrakce funkcionálního kódu. Druhý důvod, proč byl zvolen jazyk Haskell,

byla má oblíbenost tohoto jazyka, protože oproti v dnešní době používaným jazykům umožňuje nahlížet na problém z jiného úhlu. A díky funkcionálnímu paradigmatu je také lehké výsledný kód zparalelizovat, čehož silně využívám v interpretu jazyka Rose.

2.1 Použité knihovny

2.1.1 Gtk2Hs

Gtk2Hs realizuje vrstvu mezi knihovnou GTK+ napsanou v C a jazykem Haskell. Gtk2hs slouží pro tvorbu grafického uživatelského prostředí.

2.1.2 SourceView

SourceView realizuje mezivrstvu mezi dodatečnou knihovnou pro GTK+ schopnou zobrazovat zdrojový kód a Haskellem. Zobrazením zdrojového kódu se myslí podpora pro číslování řádků, barvení kódu, autocompleter atp.

2.1.3 Parsec

Knihovna čistě napsaná v Haskellu. Slouží pro parsování textu. Parsec podporuje rekurzivní parsování, to znamená, že syntaxe parsovaného textu může používat rekurzivně zanořovatelné výrazy a konstrukty.

2.2 Použité aplikace při tvorbě IDE

2.2.1 GHC

Interaktivní kompiler Haskellu. Použit pro sestavení mé práce. Díky módu REPLu bylo ušetřeno mnoho času.

2.2.2 Gedit

Prostý editor textu se zvýrazňováním syntaxe Haskellu, číslováním řádků a zvýrazňováním závorek. Použit pro editaci kódu mého projektu.

2.2.3 Glade

Grafický návrhář GTK+ rozhraní. Byl použit pro navržení GUI.

3 Vnitřní struktura aplikace

Pro pochopení vnitřní struktury vývojového prostředí určeného pro kernelový vývoj je nutno napřed ozřejmit odlišnosti od klasických IDE určených pro vývoj user space aplikací. Níže uvedené postupy sestavovacího procesu počítají s kompilovanými programy, nikoli interpretovanými ze zdrojového kódu či bajtkódu.

3.1 Sestavovací proces

3.1.1 Postup při sestavování user space aplikace

- Kompilace zdrojových kódů do objektových souborů.
- Slinkování objektových souborů a připojení standardní knihovny jazyka. Pro slinkování je použit linker skript popisujících strukturu binárního vykonavatelného souboru, do kterého kompilujeme program. Systém Windows používá formát EXE, Unixové systémy používají obvykle formát ELF.
- Hotová a spustitelná aplikace operačním systémem.

3.1.2 Postup při sestavování systémové aplikace

- Kompilace zdrojových kódů do objektových souborů.
- Slinkování objektových souborů. Není připojena standardní knihovna jazyka,

protože nově vytvářený systém nemůže použít API operačního systému. Na čemž závisí většina funkcí v STD knihovně. Linker pro slinkování použije námi vytvořený linker skript popisující binární formát našeho kernelu. To je první odlišnost mezi sestavovacím procesem kernelu a user space aplikace. IDE musí mít možnost zadat linkeru uživatelem vytvořený linker skript.

- Připojení výsledného binárního souboru k jiným binárním souborům v případě vícefázového binárního souboru. Vícefázovým binárním souborem je myšlen program, který vznikne ze dvou či více binárních souborů navzájem provázaných (a spojených) jen svým umístěním vzhledem k sobě. Jako příklad uvedu bootloadery. Každý bootloader v dnešní době je více fázový. Stejně tak je více fázový program, který instaluje OS na PC. Někdy bývají více fázové i kernely (a viry, to jen pro zajímavost).
- Hotový binární soubor spustitelný při bootování PC či v emulátoru architektury.

3.1.3 Rekapitulace

Z výše popsaných postupů sestavování user space aplikací a systémových aplikací je vidět, že IDE specializované pro kernelový vývoj musí umožňovat učinit velké zásahy do sestavovacího procesu. Musí umožnit vložení vlastního linker skriptu do kompilačního procesu a umožnit připojení výsledného binárního souboru k jinému binárnímu souboru. A nakonec musí místo klasického terminálu výsledný soubor pouštět v emulátoru architektury, pro který daný kernel vyvíjíme. Díky vysoké variabilitě možných nastavení kompilačního procesu kernelu je obtížné vytvořit pohodlně použitelné ovládací rozhraní. Z toho důvodu byly pro účely se sestavování binárních souborů vytvořeny specializované systémy, jako například CMAKE, jejichž nevýhoda dlí především v jejich komplexnosti. Mnoho IDE pro C/C++ interně CMAKE používá, takže při troše snahy lze kernel vyvíjet v jakémkoli IDE umožňujícím zasáhnout hlouběji do procesu kompilace. Daný zásah ale nebývá vždy moc jednoduchý a leckdy ani není možný.

3.2 Komponenty DKIDE

DKIDE se skládá ze tří hlavních komponent. Datového modelu projektu, uživatelského rozhraní a interpretu mnou vytvořeného jazyka Rose.

3.2.1 Datový model projektu

Model v sobě obsahuje globální stav celého IDE. Protože GTK2HS knihovna realizuje pouze tenký wrapper nad knihovnou GTK+ napsanou v imperativním jazyce C, je třeba pro zajištění udržení globálního stavu projektu potlačit neměnnost dat a učinit objekt, ve kterém je uložen stav měnitelný, to je provedeno pomocí měnitelné reference IORef. Tím je potlačena implicitní neměnnost dat v Haskellu. Pokud by byla použita knihovna implementující funkcionálně reaktivní styl event driven kódu, bylo by možné neměnitelnost zachovat. Datový model projektu vyjádřený ve zdrojovém kódu Haskellu vypadá následovně:

```
data Project = Project {
        projectName :: String,
        projectPath :: String,
        projectFiles :: [String],
        libraryFiles :: [Tree String],
        currentOpenFile :: [Int],
        gppArgs :: [String],
        ldArgs :: [String],
        qemuArgs :: [String],
        ideArgs :: ML.Map String String
} deriving(Eq,Show,Read)
```

projectName

Obsahuje jméno aktuálně otevřeného projektu.

projectPath

Obsahuje plnou cestu za adresářem, v němž je uložen projekt.

projectFiles

Obsahuje seznam souborů projektu. Udáváno je pouze jméno, projectPath slouží jako prefix.

libraryFiles

Obsahuje seznam stromů. Přičemž každý kořen stromu obsahuje prefix cesty pro jména souborů v jeho listech.

currentOpenFile

Obsahuje směrová data ukazující na aktuálně otevřený soubor v IDE. Pokud má seznam jen jeden prvek, obsahuje index do seznamu projectFiles. Pokud má dva prvky, první index ukazuje do seznamu libraryFiles a druhý index ukazuje na konkrétní list stromu.

gppArgs

Obsahuje seznam parametrů kompileru G++.

ldArgs

Obsahuje seznam parametrů linkeru LD.

qemuArgs

Obsahuje seznam parametrů emulátoru QEMU. V aktuální verzi IDE se nepoužívá. Necháno z důvodu nutnosti přepsat mnoho kódu při odstranění.

IdeArgs

Obsahuje asociativní kontejner ukládající parametry IDE. Díky tomuto datovému členu je možné datový model DKIDE takřka neomezeně rozšiřovat bez nutnosti měnit stávající zdrojový kód.

3.2.2 Grafické uživatelské rozhraní

Grafické uživatelské rozhraní je velmi strohé a jednoduché. Neobsahuje v sobě ani jeden ovládací prvek, obsahuje pouze vizualizační prvky pro text a strukturu projektu a editační prvky pro text. GUI bylo vytvořeno pomocí nástroje GLADE, který navržené rozhraní převede do XML kódu, z kterého poté dokáže knihovna GTK+ vyextrahovat a vytvořit objekty, s kterými poté lze v kódu pohodlně pracovat bez nutnosti jejich inicializace.

3.2.2.1 Zobrazení struktury projektu

Realizováno pomocí widgetu TreeView (gtk_tree_view). Jedná se o jediný Widget v IDE, který je schopen zasahovat do struktury projektu (nikoli ale do jeho nastavení). Na onClick event je připojen kód, který se stará o přidávání a odebírání souborů a knihoven z projektu. Sice bylo možné tyto operace přenést do REPLu, ukázalo se však, že se jedná o zbytečné zdržování uživatele.

3.2.2.2 Editor kódu

Realizováno pomocí widgetu SourceView. V tuto chvíli je k dispozici barvení kódu, zvýrazňování párů závorek, číslování řádků a zvýrazňování aktuálního řádku. Doplňování textu ještě není implementováno a bude přidáno v dalších verzích IDE.

3.2.2.3 REPL interpretu jazyka Rose

Popsáno níže v sekci pojednávající o interpretu.

3.2.3 Interpret jazyka Rose

Jedná se o nejdůležitější a algoritmicky nejsložitější část DKIDE. Přes REPL jazyka Rose se ovládá celé IDE, což je důvod, proč nejsou třeba žádné grafické ovládací prvky jako v jiných vývojových prostředích. Interpret se skládá z níže popsaných komponent.

3.2.3.1 Datový model interpretu

Hodnotové datové typy jsou kontejnery pro primitivní datové typy jazyka Rose. Datové typy výrazů jsou kontejnery, které v sobě ukládají logiku kódu, tvoří abstraktní syntaktický strom programu, který poté evaluátor redukuje (interpretuje) na primitivní datové typy jazyka Rose.

3.2.3.1.1 Datové typy hodnot

Níže je kód datové realizace kontejneru s primitivními datovými typy jazyka Rose.

```
data RVal =
        RBool Bool              |
        RInt Int                |
        RFloat Double           |
        RChar Char              |
        RUnit                   |
        RTuple [RVal]           |
        RList [RVal]            |
        RMap [(RVal,RVal)]      |
        RLambda [String] RExpr (ML.Map String RVal)
                deriving(Show,Read,Eq)
```

RBool

Wrapper pro primitivní datový typ Bool.

RInt

Wrapper pro primitivní datový typ signed 64 bit Int.

RFloat

Wrapper pro primitivní datový typ signed double precision Float.

RChar

Wrapper pro primitivní datový typ unicode Char.

RUnit

Wrapper pro primitivní datový typ Unit, který funguje jako bezvýznamová hodnota, užitečná ve chvíli, když funkce realizující interakci se světem nepotřebuje vrátit žádnou hodnotu, ale syntaxe (a evaluátor) jazyka vyžaduje, aby každá funkce vracela návratovou hodnotu. Taktéž lze s typem Unit indikovat selhání funkce. Díky RTTI (run time type informations) je poté možno chyby v běžícím kódu detekovat.

RTuple

Wrapper pro primitivní datový typ Tuple.

RList

Wrapper pro primitivní datový typ List.

RMap

Wrapper pro primitivní datový typ Map. Map je key-value asociativní kontejner. V tuto chvíli je řešen jako prostý list. Takže efektivita kontejneru je O (n). Toto omezení je způsobeno tím, že není možné vložit do datového typu Data.Map, jenž obsahuje std knihovna Haskellu, datový typ RVal, protože jako celek nepodporuje porovnávací operace, pouze některé jeho konstruktory. O(N log N) bude přidána s další verzí IDE. Zdržení je dáno nutností implementovat vlastní vyhledávací strom.

RLambda

Wrapper pro primitivní datový typ Lambda. RLambda je kontejner pro abstraktní syntaktický strom kódu funkce. V interpretu neexistuje žádný objekt realizující pojmenovanou funkci. Každá funkce je nepojmenovaná, pokud je třeba, aby funkce měla přiřazeno jméno, je k ní připojeno pomocí ntice.

3.2.3.1.2 Datové typy výrazů

Jedná se o objekty, které jsou datovými protějšky výrazů ve zdrojovém kódu.

```
data RExpr =
        Literal RVal                          |
        Var String REPos                      |
        Call RExpr [RExpr] REPos              |
        LocalBinds [(String,RVal)] RExpr      |
        Case RExpr [(RExpr,RExpr)] REPos      deriving(Show,Read,Eq)
```

Literal

Realizuje napevno udanou primitivní datovou hodnotu jazyka Rose v kódu. Každá primitivní datová hodnota má tu vlastnost, že se vyhodnotí sama do sebe.

Var

Realizuje zpřístupnění proměnné (z vnějšího rozsahu platnosti) v kódu.

Call

Realizuje funkční volání.

LocalBinds

Realizuje rozšíření úrovně rozsahu platnosti pomocí lokálních definic funkcí a proměnných, které jsou neměnné. Neměnitelné proměnné zní sice protichůdně, je to ale nejlepší způsob, jak daný stav popsat. Po vytvoření je jméno napevno spojeno s jednou hodnotou, dané jméno ale může být vytvořeno vícekrát, takže hodnota se mění, ale v jedné konkrétní úrovni rozsahu platnosti ne. Takže je splněna podmínka neměnnosti (a tím pádem nepotřebnosti zamykání paměti).

Case

Realizuje výraz v jiných jazycích nazývaný switch. V jazyce Rose se ale jedná o mírně vylepšený switch. Pomocí Case je nahrazeno i klasické větvení pomocí If, proto zde není žádný objekt realizující výraz větvení If.

3.2.3.2 Parser

Jedná se o důležitý a implementačně složitý modul, protože převádí zdrojový kód do datového modelu interpretu, tj. do abstraktního syntaktického stromu kódu. Složitost je o to větší, že jazyk Rose je celý složen z výrazů, které se mohou rekurzivně vzájemně do sebe nekonečně zanořovat. Z toho důvodu musí být parser silně rekurzivní. Díky funkcionální parsovací knihovně parsec ale nebyl tento problém tak závažný, jak by byl v případě použití libovolného imperativního jazyka.

3.2.3.3 Serializer

Taktéž velmi důležitý modul. Stará se o opačný úkol než parser. To znamená, že převádí abstraktní syntaktický strom na kód. Toho se využívá při zobrazování instancí primitivních datových typů jazyka Rose včetně funkcí. REPL a zabudovaná funkce show silně využívá tohoto modulu. Díky rekurzi nebylo těžké tento modul implementovat. Pokud by místo rekurze ale bylo využito imperativních iteračních cyklů, implementovat serializer by bylo skutečně obtížné.

3.2.3.4 Evaluátor

Srdce interpretu. Evaluátor transformuje abstraktní syntaktický strom kódu do primitivních datových typů jazyka, které lze poté zobrazit na výstupu. Slovo transformace místo interpretace se obvykle neužívá, dokonale ale popisuje činnost evaluátoru. Opět díky rekurzi není implementace evaluátoru přespříliš náročná tak, jak by mohla být, kdyby bylo využito iteračních cyklů. Nevýhodou ale je, že rekurze má nižší výkon z důvodu pomalých funkčních volání na architektuře x86 a nutnosti alokovat dodatečnou paměť pro funkční zásobník. Evaluátor jazyka Rose podporuje paralelní interpretaci výrazů ve stejné hladině abstraktního syntaktického stromu, přičemž není nutné paměť díky persistenci dat zamykat.

3.2.3.5 Generický REPL

Jedná se o modul, který realizuje univerzální REPL pro interpret. REPL neboli Read Eval Print Loop. Přečti, vyhodnoť, vytiskni a opakuj. Modul je tvořen jednou funkcí, která realizuje funkci REPLu a pro vstup a výstup využívá poslaných funkcí. Dále posílá díky zamykatelné měnitelné datové struktuře volajícímu jména, která obsahuje exekuční prostředí interpretu pro realizaci automatického doplňování jmen. Parsování interpretovaných souborů probíhá v REPLu paralelně, čímž je výrazně zvýšen výkon.

3.2.3.6 Grafický REPL

Tvoří mezivrstvu mezi GTK widgetem TextView a generickým REPLem. Upravuje TextView tak, aby se choval jako REPL. To znamená, výstup generického REPLu vkládá do TextView a po stisknutí klávesy enter vkládá zadaný text do zamykatelné fronty, z které načítá vstup generický REPL. Protože TextView nebyl koncipovaný pro zadávání textu po stisknutí entru, neboť původně sloužil pro editaci textu, nikoli pro zadávání kratšího textu jako LineEdit, bylo tuto schopnost nutno implementovat.
Dále je TextView upraven tak, že neumožňuje vymazat pomocí backspace dříve zadaný obsah generickým REPLem. Stejně tak bylo nutno implementovat zablokování zadávání textu do TextView na místě dříve zadaného obsahu generickým REPLem. Pro pohodlnější užívání byla přidána historie zadaných výrazů, zpřístupnitelná pomocí

šipek nahoru a dolu. Taktéž bylo přidáno automatické doplňování jmen pomocí tabulátoru, které je realizováno díky měnitelné referenci na list jmen z exekučního prostředí, které exportuje generický REPL. Všechny změny a dodané dovednosti vyžadovaly silných zásahů do handlerů eventů widgetu. Ve výsledku funguje tento modul tak, že vezme TextView widget, přepíše a upraví jeho handlery eventů a spojí ho s generickým REPLem, přičemž samotný generický REPL běží z důvodu zajištění plynulého běhu GUI ve vlastním vlákně.

4 Dokumentace

4.1 K synchronizaci projektu s jeho obrazem na HDD

Kompletně celý stav projektu je při každé změně synchronizován na disk. Od prostého ukládání souborů při změně až po stav rozbalení projektového stromu. IDE, můžeme zavřít a otevřít a nepoznáme jedinou změnu, vše je ve stejném stavu jako před vypnutím. Díky tomu se není třeba starat o časté ukládání práce a ani hledat ovládací prvky pro ukládání projektu.

4.2 Ke GUI

Jediným místem v GUI, které je třeba zdokumentovat, je strom projektu. Po kliknutí pravým tlačítkem myši na kořenový uzel stromu vyskočí nabídka na otevření nového projektu, založení nového projektu, připojení další knihovny či založení dalšího souboru. Použití těchto voleb je snadné a není třeba je blíže popisovat. Při zakládání nových projektů se samozřejmě žádná práce neztratí, IDE vše uloží samo. Pokud klikneme pravým tlačítkem na jméno projektového souboru, zobrazí se možnost soubor smazat. Stejná funkce se nabídne, když klikneme na jméno vložené knihovny.

4.3 K interpretu jazyka Rose

Pomocí šipek můžeme procházet historií zadaných výrazů do REPLu. Pomocí tabulátoru můžeme automaticky doplňovat napsaná jména. Interpret lze ovládat pomocí velmi omezené sady příkazů. Každý příkaz začíná dvojtečkou. Jedná se o rozlišení mezi jmény v jazyce Rose a příkazy interpretu. Protože v jazyce Rose jsou všechna jména začínající na dvojtečku rezervována pro interní použití, nemůže vzniknout záměna.

:help
zobrazí nápovědu REPLu.
:h
To samé co :help.
:load filepath
Nahraje do interpretu zdrojový soubor Rose.
:l filepath
To samé co :load.
:reload
Znovu nahraje všechny nahrané soubory v interpretu.
:r
To samé co :reload.
:clear
Vyčistí obsah REPLu.
:c
To samé co :clear.

4.4 K jazyku Rose

Mnou vytvořený jazyk Rose je minimalistický funkcionální silně dynamicky typovaný jazyk se striktním evaluačním modelem a kompletně bez výjimky persistentními daty. Jeho hlavním charakteristickým prvkem je absence jakéhokoli klíčového slova. Syntaxe jazyka je složena jen ze znaků. Jazyk má v sobě zabudovanou podporu pro paralelismus, přičemž díky persistentním datům odpadá problém se synchronizací dat a z toho plynoucími bugy a zpomalením běhu programu. Doporučuji programovat paralelní algoritmy všude, kde to jen jde. Interpret je totiž v aktuální verzi velmi

pomalý, ale díky využití všech jader procesoru je možno dosáhnout použitelné rychlosti. Rose programy běžící na jednom jádře bývají takřka nepoužitelné, co se týče rychlosti, pokud analyzují data v řádech megabajtů. Jazyk Rose není referenčně transparentní, to znamená, že v jeho rutinách je možno ovlivňovat globální stav programu. Díky tomu není možné považovat jazyk Rose za plně funkcionální. Díky absenci referenční transparentnosti funkcí si musíme při paralelizování kódu dávat pozor na případné konflikty vláken o sdílené zdroje. Vzhledem k aktuálnímu stavu API jazyka se může jednat pouze o konflikt při přístupu k souboru či k terminálu.

Metody, jak do programu vložit paralelní exekuci, jsou vysvětleny v sekci pojednávající o knihovně jazyka Rose.

4.4.1 Primitivní datové typy

Jazyk Rose podporuje devět primitivních datových typů. Za jménem datového typu následuje podoba tvorby instance daného datového typu. Tvorba instance se počítá jako výraz. Za symbol expr lze dosadit libovolný výraz podporovaný jazykem Rose.

Bool **True | False**

Bool nabývá dvou hodnot. True a False. V jazyce Rose to není jinak.

Int **{1 .. 9}+**

Int ukládá 64 bitové číslo se znaménkem.

Float **{1 .. 9}+.{1 .. 9}**

Float ukládá double presicion číslo se znaménkem.

Char **'{a .. z}'**

Char ukládá jeden znak unicode, 32 bitů.

Unit **_**

Jedná se o datový reprezentující absenci hodnoty. Instance tohoto datového typu se používá jako návratová hodnota funkcí, které nemají co vrátit. Obvykle se jedná o funkce provádějící IO.

Tuple **({expr,})**

Tuple je heterogenní datový kontejner ukládající sekvenci instancí odlišných datových typů.

List **[{expr,}]**

List je homogenní kontejner ukládající sekvenci instancí jednoho datového typu.

String **“řetězec ...“**

String je List obsahující znaky ([Char]). Ničím se od klasického sezamu neliší, ale má vlastní syntaxi tvorby instancí pro snazší používání. Na instanci datového typu String lze použít jakoukoli funkci kompatibilní se seznamy.

Map **[{expr:expr,}]**

Map je asociativní homogenní kontejner. Ukládá v sobě páry klíč-hodnota.

Lambda **\({arg,}) = expr;**

Lambda je nepojmenovaná funkce.

4.4.2 Funkce

Tvorba funkce je jediný syntaktický konstrukt, který podporuje jazyk Rose. Vše ostatní jsou výrazy.

jméno-funkce(jméno-argumentu-0, ..., ...) = expr;

Jméno funkce i jméno argumentu nesmí začínat na dvojtečku, jinak není rezervován žádný znak s výjimkou všech typů závorek. Za expr lze dosadit jakýkoli výraz podporovaný jazykem.

4.4.3 Výrazy

Srdcem jazyka Rose (a každého jiného funkcionálního jazyka) jsou výrazy. Výrazy lze do sebe donekonečna zanořovat a je garantováno, že každý výraz má svou návratovou hodnotu.

4.4.3.1 Literály

Literál je výraz tvořící datovou instanci libovolného datového typu. Jejich podoba je k nalezení v sekci primitivní datové typy.

4.4.3.2 Jmenné přístupy

Tento výraz je vytvořen pouhým napsáním jména, které je nalezitelné ve jmenném prostoru daného výrazu. Návratová hodnota tohoto výrazu je hodnota svázaná se jménem, které jsme udali.

4.4.3.3 Volání funkcí

Tento výraz realizuje volání funkce s udanými argumenty. Návratovou hodnotou výrazu je návratová hodnota volané funkce.

jméno-funkce(expr, ..., ...,)
Na místě argumentů může být umístěn libovolný výraz jazyka Rose.

4.4.3.4 Local binds

Tento výraz rozšiřuje jmenný prostor všech zanořených výrazů. Slouží pro lokální definice funkcí a konstant.

{
jméno-funkce(jméno-argumentu-0, ..., ...) = expr;
jméno-konstanty = výraz;
} expr
Všechny jmenné přístupy obsažené v hlavním výrazu local binds mohou přistupovat k lokálně definovaným jménům funkcí a konstant. Se zbytkem programu se nově definovaná jména nedostanou do konfliktu. Pokud nastane kolize stejných jmen, je vždy preferováno naposledy definované jméno. Výraz definující konstantu nesmí využívat jmen konstant, které jsou definovány později v local binds.

4.4.3.5 Case

Tento výraz realizuje větvení exekuce kódu. Lze použít jako klasické IF či SWITCH větvení známé z jiných jazyků.

výraz {
výraz : výraz;
...
_ : výraz;
}

Jádrem výrazu Case je seznam párů výrazů oddělených dvojtečkou. Pokud se hodnota kořenového výrazu Case rovná hodnotě výrazu před dvojtečkou, je vykonáván výraz za dvojtečkou a Case výraz vrátí jeho návratovou hodnotu. Pokud se kořenový výraz nerovná hodnotě žádné větvě, je vyvolána exception. Větev, jejíž první výraz je nahrazen _, je větev defaultní, která se vykoná bez testu na rovnost.

4.5 K rozhraní jazyka Rose

4.5.1 Jak číst typové signatury funkcí

Int|Float|a
Datové typy oddělené svislou čárou znamenají, že na daném místě můžou být instance takto oddělených typů. Malá písmena slouží pro označení jakéhokoli datového typu. V jedné typové signatuře může ale jedno konkrétní písmeno stát pouze pro jeden konkrétní datový typ.
a, ...
Datový typ, čárka a tři tečky slouží pro označení neomezeného počtu argumentů daného datového typu.
Datový-typ <- jméno-funkce(x0#argument-0, ...)
Na levé straně je udán návratový datový typ funkce. Argument, před kterým je # a nějaký znak či číslo, znamená pojmenovaný argument, který bude sloužit v popisku funkce jako reference na daný parametr. Při použití dané funkce v interpretu se žádné # nepíší.

4.5.2 Zabudované funkce jazyka

Zabudované funkce jsou exekučním jádrem jazyka, z kterého se staví vše ostatní. Nemusí dodržovat pravidla, která platí pro obyčejné funkce. Aktuální verze interpretu používá zabudované funkce s měnitelnými počty argumentů, což je vlastnost, kterou klasické funkce nemají.

Int|Float <- add(Int|Float, ...)

Sečte neomezené množství hodnot ve vstupních argumentech.

Int|Float <- sub(Int|Float, ...)

Odečte neomezené množství hodnot ve vstupních argumentech.

Int|Float <- mul(Int|Float, ...)

Vynásobí neomezené množství hodnot ve vstupních argumentech.

Int|Float <- div(Int|Float, ...)

Vydělí neomezené množství hodnot ve vstupních argumentech.

Int|Float <- rem(Int|Float, ...)

Vydělí se zbytkem neomezené množství hodnot ve vstupních argumentech.

Int|Unit <- to-int(Float|Char|Bool|String)

Převede dané datové typy na Int. Číslo ve stringu může být udáno v 10, 16 i 2 soustavě. V případě selhání konverze vrací Unit.

Float|Unit <- to-float(Int|Char|Bool|String)

Převede dané datové typy na Float. Číslo ve stringu může být udáno pouze v desítkové soustavě. V případě selhání konverze vrací Unit.

Bool <- to-bool(Int|Float)

Převede dané datové typy na Bool.

Char <- to-char(Int)

Převede Int na Char.

String <- to-bin(Int)

Převede Int na číslo ve stringu o základu 2.

String <- to-hex(Int)

Převede Int na číslo ve stringu o základu 16.

Bool <- and(Bool, ...)

Provede logickou operaci AND nad všemi argumenty a vrátí výsledek.

Bool <- or(Bool, ...)

Provede logickou operaci OR nad všemi argumenty a vrátí výsledek.

Bool <- not(Bool)

Provede negaci vstupu.

Bool <- equal(a,b)

Porovná dvě datové instance, pokud jsou shodné, vrátí True, v opačném případě False.

Bool <- nequal(a,b)

Porovná dvě datové instance, pokud jsou shodné, vrátí False, v opačném případě True.

Bool <- greater(a,b)

Provádí logickou operaci a > b.

Bool <- lesser(a,b)

Provádí logickou operaci a < b.

Bool <- eqorgr(a,b)

Provádí logickou operaci a >= b.

Bool <- eqorle(a,b)

Provádí logickou operaci a <= b.

a <- head([a])

Vrací první prvek seznamu. Náročnost operace je O(1).

[a] <- tail([a])

Vrací všechny prvky seznamy vyjma prvního. Náročnost operace je O(1).

[a] <- init([a])

Vrací všechny prvky seznamy vyjma posledního. Náročnost operace je O(n - 1).

a <- back([a])

Vrací poslední prvek seznamu. Náročnost operace je O(n - 1).

[a] <- append(1#[a],2#[a])

Spojí seznam 1 a 2. Náročnost operace je O(length(1)).

[a] <- cons(1#a,2#[a])

Přidá prvek 1 na začátek seznamu 2. Náročnost operace je O(1).

[a] <- repeat(v#a,n#Int)

Vytvoří N prvkový list naplněný hodnotami v.

Int <- len([a])

Vrátí délku seznamu. Náročnost operace je O(n).

a <- at([a],n#Int)

Vrátí prvek seznamu na konkrétním umístění. Indexy začínají od 0. Náročnost operace je O(n).

Bool <- member(k#a,n#[a:b])

Vrací True, když je klíč k obsažen v kontejneru n. Náročnost operace je O(length(n)). Zatím neefektivní operace z důvodu primitivní implementace kontejneru.

b <- lookup(k#a,n#[a:b])

Najde hodnotu asociovanou s klíčem k v kontejneru n. Pokud klíč v kontejneru není, je vyvolána exception. Náročnost operace je O(length(n)). Zatím neefektivní operace z důvodu primitivní implementace kontejneru.

[a:b] <- insert(k#a,v#b,n#[a:b])

Vloží pár klíč hodnota (k, v) do kontejneru n. Pokud je již klíč v kontejneru, je přepsána hodnota novou hodnotou. Náročnost operace je O(length(n)). Zatím neefektivní operace z důvodu primitivní implementace kontejneru.

[a:b] <- remove(k#a,n#[a:b])

Odstraní klíč k z kontejneru n. Náročnost operace je O(length(n)). Zatím neefektivní operace z důvodu primitivní implementace kontejneru.

Unit <- seq(a, ...)

Vyhodnotí sekvenčně všechny argumenty a vrátí Unit. Slouží pro realizování IO operací.

(a|..., ...) <- par(a|..., ...)

Vyhodnotí paralelně všechny argumenty libovolných typů a vrátí je v jedné ntici. Není třeba paměť zamykat, protože je neměnná. Velmi silný nástroj, z kterého lze vytvořit jakýkoli funkcionální paralelní algoritmus.

String <- type-info(a)

Vrátí datový typ poslaného argumentu. Datové typy jsou uvedeny s prefixem „R“.

a <- type-check(a)

Zkontroluje typovou korektnost poslané datové instance. Ve chvíli, kdy najde chybu, vyvolá exception. Zkontrolovat lze jakoukoli datovou instanci, jediný zdroj chyb ale může být v seznamech. Funkce pracující se seznamy totiž neprovádí plnou typovou kontrolu, protože je příliš výpočetně náročná. Z toho důvodu se může stát, že se do seznamu vloudí i datový typ, který není ve shodě se zbytkem prvků seznamu.

Unit <- error(String)

Vyvolá vyjímku nesoucí námi poslanou zprávu.

a <- trace(m#String,2#a)

Pošle na výstup zprávu m a vrátí hodnotu argumentu 2.

String <- show(a)

Zobrazí textovou reprezentace jakékoli datové instance použitelné v jazyce Rose.

a <- read(String)

Převede z textové reprezentace libovolný datový typ do datové instance. Pokud nejde poslaný řetězec převést na datovou instanci, vrátí read Unit.

Unit <- put-str(String)

Vypíše na výstup poslaný řetězec.

Unit <- put-str-ln(String)

Vypíše na výstup poslaný řetězec a řádkový zlom.

String <- read-ln()

Načte ze vstupu řetězec až po zadání entru.

String <- read-all()

Načte ze vstupu všechny znaky. Hodí se při tvorbě jednoúčelových konzolových programů.

Unit <- write-file(n#String,c#String)

Zapíše do souboru n řetězec c. Pokud soubor něco obsahoval, je to přepsáno. Pokud soubor neexistoval, nastane pád interpretu.

String <- read-file(String)

Načte obsah souboru. Pokud soubor neexistuje, spadne interpret. Opět se jedná o bug.

Unit <- append-files(String, ...)

Spojí všechny poslané soubory do jednoho. Když libovolný soubor neexistuje, interpret spadne. Opět bug.

Bool <- file-exist(String)

Otestuje, zdali existuje daný soubor.

Unit <- delete-files(String, ...)

Smaže všechny udané soubory. Neexistence souborů funkci nevadí.

4.5.3 Standardní knihovna

Standardní knihovna je zatím malá, ale kompletně implementuje funkcionální rozhraní nad seznamy.

[b] <- map(b <- \(a),[a])
Aplikuje poslanou funkci na každý prvek poslaného seznamu a vrátí jeho transformovanou verzi.
[b] <- mapi(b <- \(a,Int),[a])
Aplikuje poslanou funkci na každý prvek poslaného seznamu a vrátí jeho transformovanou verzi. Poslaná funkce jako druhý argument bere index právě zpracovávaného prvku.
[b] <- pmap(b <- \(a),[a])
Funguje stejně jako funkce map s tím rozdílem, že každý prvek zpracovává paralelně. Vhodné použít pokud mapovaná funkce dělá výpočetně náročnější úkol než je cena vytvoření paralelní exekuce. Tato cena je naštěstí díky Haskellu velmi malá, a interpret pomalí, takže zparalelizování jakékoli mapované funkce vede ke zrychlení chodu aplikace.
[b] <- pmapi(b <- \(a,Int),[a])
Funguje stejně jako funkce pmap s tím rozdílem, že posílá funkci index zpracovávaného prvku.
[a] <- filter(Bool <- \(a),[a])
Na každý prvek poslaného seznamu aplikuje poslaný predikát, prvky, na které vrátí predikát False, jsou ze seznamu odstraněny.
b <- foldl(b <- \(b,a),acc#b,[a])
Provádí minimalizaci listu. Poslaná funkce bere jako první argument akumulátor a jako druhý argument prvek listu. Výsledná hodnota poslané funkce se ukládá do akumulátoru. Po projití všech prvků listu je vrácen akumulátor jako návratová hodnota funkce foldl. Funkce foldl prochází prvky zleva doprava (od prvního prvku do posledního).

a <- foldl1(a <- \(a,a),[a])

Stejné jako funkce foldl s výjimkou toho, že datový typ akumulátoru musí být stejný jako datový typ prvků zpracovávaného seznamu. Akumulátor je inicializován prvním prvkem seznamu. Pokud je poslán prázdný seznam, je vyvolána exception.

b <- foldr(b <- \(b,a),acc#b,[a])

To samé co funkce foldl s výjimkou pořadí zpracování prvků. Prvky se zpracovávají zprava do leva. Neboli od posledního prvku k prvnímu.

a <- foldr1(a <- \(a,a),[a])

To samé, co funkce foldl1, s výjimkou pořadí zpracování prvků. Zpracovávají se od posledního do prvního.

[a] <- concat([[a]])

Spojí seznam seznamů na vstupu do jednoho seznamu.

[Int] <- range(from#Int,to#Int,step#Int)

Vygeneruje seznam s čísly od from až do to, přičemž čísla se inkrementují či dekrementují o hodnotu step. Hodnota step je vždy kladná. Pokud chceme, aby byly kroky záporné, otočíme hranice intervalu from to.

[Int] <- srange(Int,Int)

To samé, co range, s výjimkou hodnoty step, která je zde napevno nastavená na 1.

[a] <- take(n#Int,[a])

Vrátí prvních N prvků poslaného seznamu. Náročnost operace je O(n).

[a] <- take-while(Bool <- \(a),[a])

Vytvoří sub seznam z poslaného seznamu, přičemž posledním prvkem vytvořeného sub seznamu je prvek ležící před prvkem, na který poslaný predikát vrátil False (to znamená, že na všechny předchozí vrátil predikát True).

String <- take-to(substr#String,str#String)

Vytvoří substring obsahující všechny znaky před prvním výskytem substr v str.

[a] <- drop(n#Int,[a])

Vrátí poslaný seznam bez prvních n prvků.

[a] <- drop-while(Bool <- \(a),[a])

Vrátí poslaný seznam bez prvních N prvků do prvního případu hodnoty False poslaného predikátu.

String <- drop-to(substr#String,str#String)

Vrátí String bez prvních N znaků do prvního objevení substr v str.

Bool <- elem(val#a,list#[a])

Vrátí True, pokud je val obsažen v listu. V opačném případě False.

Bool <- search(substr#String,str#String)

Vrátí True, když je obsažen substr v str, jinak False.

[String] <- str-slice(substr#String,str#String)

Rozkrájí str na množinu seznamů. Oddělovač je substr.

[(a,b)] <- zip([a],[b])

Sloučí dva seznamy do jednoho pomocí ntice.

([a],[b]) <- unzip([(a,b)])

Rozloží seznam ntic do dvou seznamů v ntici.

[c] <- zip-with(c <- \(a,b),[a],[b])

Sloučí dva seznamy do jednoho pomocí poslané funkce. Výsledný seznam je krátký jako nejkratší ze slučovaných seznamů.

[d] <- zip-with(d <- \(a,b,c),[a],[b],[c])

To samé co zip-with, jen se třemi seznamy.

[a] <- intersperse(v#a,[a])

Vloží prvek v mezi každý prvek v poslaném seznamu.

([a],[a]) <- partition(Bool <- \(a),[a])

Rozdělí seznam na dva v ntici. Prvky, na které vrátí poslaný predikát True, se vloží do prvního seznamu, zbytek do druhého.

[a] <- sort([a])

Seřadí prvky od nejmenšího po největší. Je využito optimalizovaného quicksortu s efektivitou

O(n log n).

Bool <- empty([a])

Pokud je poslaný seznam prázdný, vrátí True, v opačném případě False.

[String] <- lines(String)

Rozseká vstupní řetězec na mnoho podřetězců. Oddělovacím znakem je řádkový zlom.

String <- unlines([String])

Seznam řetězců spojí a proloží je řádkovými zlomy.

Int <- inc(Int)

Inkrementuje o 1 poslané číslo.

Int <- dec(Int)

Dekrementuje o 1 poslané číslo.

Int|Float <- abs(Int|Float)

Vrátí absolutní hodnotu poslaného čísla.

Int|Float <- sum([Int|Float])

Sečte čísla v poslaném seznamu.

a <- min([a])

Vrátí nejmenší prvek v poslaném seznamu.

a <- max([a])

Vrátí největší prvek v poslaném seznamu.

c <- \(b,a) <- flip(c <- \(a,b))
Prohodí argumenty poslané funkce a vrátí ji.

a <- reflux(Unit <- \(a),a)

Vrátí poslanou hodnotu beze změny, ale předtím jí ještě pošle námi poslané funkci.

a <- id(a)

Vrátí beze změny poslanou hodnotu.

Unit <- print(a)

Vypíše na výstup poslanou hodnotu.

Unit <- print-ln(a)

Vypíše na výstup poslanou hodnotu a řádkový zlom.

4.5.4 DKIDE API

Jedná se o zabudované funkce speciálně vytvořené pro ovládání IDE. Díky nim lze projekt sestavit, spustit či s ním jinak libovolně pracovat podle našich potřeb. Opět jediným omezením, které musíme mít na paměti, je malý výkon interpretu, takže nedoporučuji dělat pomocí jazyka Rose analýzu zdrojových kódů. Každá funkce pracující nějakým způsobem s IDE začíná na prefix „dk“.

Unit <- dk-help()

Zobrazí grafickou nápovědu k IDE.

String <- dk-path-sep()

Vrátí oddělovací znak použitý v systému při oddělovaní jmen v cestě za souborem.

String <- dk-base-dir()

Vrátí cestu za složkou otevřeného projektu.

String <- dk-proj-name()

Vrátí jméno projektu.

[String] <- dk-lib-files()

Vrátí seznam všech souborů knihoven. Cesty jsou uvedeny v absolutním formátu.

[String] <- dk-proj-files()

Vrátí seznam všech souborů projektu. Cesty jsou uvedeny v absolutním formátu.

(report#String|Unit,path#String|Unit) <- dk-compile(String)

Zkompiluje udaný soubor. Při kompilaci je využito G++ argumentů nastavených dříve. Tato funkce zkompiluje C++ [.cpp], C [.c] i S [.s] soubory. Pokud nesouhlasí typ souboru, vrátí funkce report jako Unit. Pokud selže kompilace z důvodu syntaktických chyb, je jako unit vrácen path (a report obsahuje zprávu kompileru). Pokud kompilace proběhne v pořádku, je path nastavena na nově vytvořený objektový soubor.

(report#String,path#String|Unit) <- dk-link(name#String,scr#String,ofiles#[String])

Slinkuje objektové soubory ofiles s využitím LD linker skriptu scr a výsledný binární soubor má jméno name. Ofiles i scr jsou zadávány v absolutní formě cesty za soubory. Name je prosté jméno. Návratový datový typ je ntice obsahující report (chybový výstup linkeru) a path (absolutní cestu za binárním souborem, pokud linkování selže, je vrácena Unit).

Unit <- dk-run(fda,fdb,hda,hdb,hdc,hdd,cd,mcount,bopt)

Spustí emulátor QEMU. Všechny argumenty jsou instance datového typu String. Pro konkrétní nastavení se podívejte na oficiální dokumentaci QEMU, jména argumentů jsou stejná jako jména argumentů QEMU.

Unit <- dk-make-lib-symlinks()

Vytvoří symlinky v kořenovém adresáři projektu na knihovny. Nutno zavolat před kompilačním procesem, jinak není možné ze souboru projektu přistupovat k souborům knihoven.

Unit <- dk-destroy-lib-symlinks

Zničí symlinky v kořenovém adresáři projektu na knihovny. Možno zavolat po kompilačním procesu s cílem vyčistit adresář, není to ale nutné.

Unit <- dk-save-current-file()

Uloží právě otevřený soubor na disk. Nutno zavolat před kompilací, aby výsledný binární soubor měl v sobě zakomponovány i nedávné změny.

Int|Unit <- dk-tab-ident(Nic|Int)

Pokud nepošleme žádný argument, vrátí velikost odsazení tabulátoru. Pokud pošleme Int, nastaví velikost odsazení na námi udanou hodnotu a vrátí Unit.

[String]|Unit <- dk-gpp-args(Nic|[String])

Pokud nepošleme této funkci žádný argument, vrátí seznam argumentů G++. Pokud pošleme argument, musí se jednat o seznam řetězců nových argumentů, kterými jsou přepsány stávající.

[String]|Unit <- dk-ld-args(Nic|[String])

Pokud nepošleme této funkci žádný argument, vrátí seznam argumentů LD. Pokud pošleme argument, musí se jednat o seznam řetězců nových argumentů, kterými jsou přepsány stávající.

Unit <- buildit()

Sestaví projekt jako jednofázový binární soubor. Jedná se o předpřipravenou kompilační funkci uloženou v souboru build.rose v každém projektu, takže je možné podle ní udělat funkci svoji.

Unit <- runit()

Spustí zkompilovaný soubor projektu v QEMU. Jedná se o předpřipravenou spouštěcí funkci uloženou v souboru build.rose v každém projektu. Takže je možné podle ní udělat funkci svoji.

5 Závěr

IDE, jehož tvorbou se zabýval můj projekt, splňuje požadavky, které kernelový vývojář klade na vývojové prostředí. Dále mnou vytvořené DKIDE nabízí nový pohled na ovládání IDE. V dnešní době neexistuje jiné IDE, které by bylo ovládáno kompletně z integrovaného REPLu (EMACS má podobnou filozofii, ale jedná se především o editor). Jediným větším problémem v současnosti je především pomalost interpretu. Tato nevýhoda je kompenzována jednoduchým a přitom silným paralelismem podporovaným jazykem Rose. V dalších verzích IDE se budu snažit co nejvíce zoptimalizovat běh interpretu a hlouběji ho provázat s IDE.

6 Zdroje

<1> Main_Page. osdev. [online]. 5.3.2015 [cit. 2015-03-05]. Dostupné z: http://wiki.osdev.org/Main_Page

<2> gtk. h*ackage.* [online]. 5.3.2015 [cit. 2015-03-05]. Dostupné z: *http://hackage.haskell.org/package/gtk*

<3> gtksourceview2. h*ackage.* [online]. 5.3.2015 [cit. 2015-03-05]. Dostupné z: *http://hackage.haskell.org/package/gtksourceview2*

<4> parsec. h*ackage.* [online]. 5.3.2015 [cit. 2015-03-05]. Dostupné z: *http://hackage.haskell.org/package/parsec*

<5> hoogle. h*askell.* [online]. 5.3.2015 [cit. 2015-03-05]. Dostupné z: *https://www.haskell.org/hoogle/*

<6> hayoo. h*ayoo.* [online]. 5.3.2015 [cit. 2015-03-05]. Dostupné z: *http://hayoo.fh-wedel.de/*

Seznam odborných výrazů

REPL – Neboli Read Eval Print Loop. Přečti, vyhodnoť, vytiskni a opakuj. Jedná se o formu interface nad interpretem jazyka. Můžeme se s touto interface setkat u mnoho interpretovaných jazyků. Například Haskell, Python, Ruby, Lisp atd.

Persistentní data – Neboli data, která se od okamžiku své tvorby nikdy nezmění.

IDE – Integrované vývojové prostředí

Referenční transparentnost – Funkce je referenčně transparentní ve chvíli, kdy jediným datovým vstupem jsou její argumenty a jejím jediným datovým výstupem je její návratová hodnota. Referenčně transparentní funkce nesmí mít svůj vlastní lokální stav a nesmí žádným způsobem ovlivňovat globální stav.

GUI – Grafické uživatelské rozhraní