# OBJ3 a LEGO jako nástroje pro specifikaci funkčního modelu[1]

**Karel Richta, Michal Valenta,** Katedra počítačů, FEL ČVUT Praha, Karlovo nám. 13, 121 35 Praha 2, e-mail: richta@cs.felk.cvut.cz, valenta@cslab.felk.cvut.cz

**Klíčová slova**: CASE, softwarové inženýrství, algebraické specifikace, teorie typů, OBJ3, LEGO, ECC.

**Abstrakt:**

Současné CASE systémy neposkytují žádné nástroje pro formální kontrolu správnosti přechodu mezi jednotlivými zjemňovacími kroky (refinement steps) při vývoji funkčního modelu systému. Každý zjemňující krok může být chápán jako úplná specifikace vyvíjeného softwarového díla (samozřejmě na dané úrovni abstrakce). Vývojář je odpovědný za to, že jednotlivé kroky (úrovně abstrakce) jsou vůči sobě navzájem korektní. V příspěvku představujeme dva velmi odlišné nástroje pro formální specifikace — OBJ3 a LEGO — které by bylo možné pro výše popsaný účel využít. Hlavní idea je ilustrována na příkladu formální specifikace a implementace problému řazení (tedy až v nejnižší vrstvě abstrakce funkčního modelu — v úrovni minispecifikací). Obecná specifikace řazení i jedna z možných implementací — Quick sort jsou vyjádřeny v OBJ3 i v notaci systému LEGO. Důkaz korektnosti zvolené implementace je proveden v systému LEGO.

## 1. ÚVOD

Obecně se při postupném zjemňování (stepwise refinement) používají dvě metody — pro funkční a pro datový model vyvíjeného systému. Současné CASE systémy podporují obvykle pouze samostatný vývoj datového a funkčního modelu, nikoli paralelní vývoj obou.[2]

Obvyklou CASE technikou pro vyjádření funkčního modelu jsou *Diagramy datových toků* (Data Flow Diagrams — dále DFD). Hierarchie DFD není specifikací v pravém slova smyslu. Spíše můžeme každou "logickou úroveň" DFD chápat jako samostatnou specifikaci systému — ovšem na patřičné úrovni abstrakce. Tedy, horní vrstvy DFD popisují pouze víceméně syntaktickou strukturu systému, zatímco sémantika je ukryta pouze v listech DFD stromu —*minispecifikacích*.

[1] Tento výzkum je částečně podporován grantem FRVŠ 23/98006/336.

[2] Zde se samozřejmě nabízejí objektově orientované metodologie, které zdůrazňují úzký vztah mezi daty a metodami. V těchto metodologiích se obvykle střídají postupy vývoje datového a funkčního modelu (například Object Model a Interface Model v metodologii FUSION). Nicméně také zde chybí jednotné pozadí, které by umožňovalo formální kontrolu správnosti přechodu mezi "logickými hladinami" metodologie — kontrola konzistence jednotlivých hladin je pouze syntaktická.

Z hlediska vývoje systému tedy nelze DFD chápat jako sadu postupných kroků přes jednotlivé úrovně DFD, kde každá úroveň může být verifikována proti libovolné jiné úrovni. Jedná se naopak o postupné zjemňování vždy kompletní specifikace. Každý následující krok je postupným zjemněním předchozího modelu — používá komplikovanější datové typy a jiné funkční detaily, které byly na předchozí úrovni skryty. Pokud je ona abstrakce každé vyšší vrstvy vůči nižší dobře popsána, lze provést formální kontrolu správnosti přechodu mezi úrovněmi.

Velmi pěknou analogii v CASE systémech jsou verze. Každá verze je kompletní specifikací, přičemž každá pozdější verze je detailnějším zpracováním verze předchozí. Při použití formálního popisu (například algebraických specifikací), lze samozřejmě konstruovat formální důkaz konzistence jednotlivých verzí.

Idea pracovat co nejdéle s formální specifikací systému je v kontextu s všeobecným trendem softwarového inženýrství: přechodem od kódem řízeného stylu práce ke specifikací řízenému stylu práce.

V tomto příspěvku se zaměříme na funkční model systému, který, na rozdíl od datového modelu[3], nenaplňuje v uspokojivé míře trend přechodu ke specifikací řízenému stylu práce. Samozřejmě, že jednou z největších příčin tohoto stavu je značná komplikovanost funkčního modelu oproti modelu datovému.

Výhody specifikací řízeného stylu práce jsou zřejmé. Takovým způsobem lze samozřejmě podporovat větší množství implementačních platforem a výrazně zvýšit využitelnost práce investované do návrhu systému. Vývojář se navíc může plně soustředit na kvalitu řešení samotného problému, aniž by byl limitován omezeními zamýšlené implementační platformy. Takový způsob práce je jistě více systematický. Další nespornou výhodou pak bude vývoj a aktualizace *"dokumentace"* (v podobě formálních specifikací různých úrovní abstrakce, které jsou ale vždy postupně konzistentní díky "dobře" popsané abstrakci a důkazům korektnosti implementace) současně s vývojem (funkčního modelu) systému prakticky "zadarmo".

## 2. VÝVOJ PO KROCÍCH

Při vývoji se obvykle uvažují čtyři modely systému: datový, funkční, dynamický a model uživatelského rozhraní. Zbytek příspěvku je věnován pouze funkčnímu modelu.

Idea vývoje po krocích (Stepwise Development) spočívá v postupném dělení problému na podproblémy a ty opět dále na podpodproblémy, dokud nedospějeme k "funkcím" (problémům), které jsou relativně snadno řešitelné. Obrazem tohoto postupu ve funkčním modelu je hierarchie DFD zakončená minispecifikacemi, které teprve popisují sémantiku řešeného problému.

---

[3] Příkladem specifikací řízeného stylu práce v datovém modelu systému je práce s ER-modelem. Po celou dobu vývoje systému pracuje návrhář na konceptuální úrovni. Transformace ER–modelu do zvolené implementační platformy během návrhu a implementace se provádí automatizovaně za přispění vývojáře.

Každý postupný krok představuje úplnou specifikaci funkčního modelu systému. První specifikace (konceptuální model) je velmi abstraktní. Každý následující krok je obohacením předchozí specifikace o podrobnosti, dokud není dosaženo úrovně minispecifikací, ze které vychází implementace. Přitom vývojář musí zaručit, že implementace je správná — tj. vyhovuje specifikaci. Na druhé straně, analytik ručí za to, že specifikace vyhovuje požadavkům uživatele.

Pokud má být postupný vývoj funkčního modelu sytému podporován nástroji pro formální specifikace, vyžaduje to, aby tyto nástroje poskytovaly dvě služby:

- *prototypování* (v ideálním případě přes celou hierarchii funkčního modelu) a
- *tvorbu důkazu*, že implementace odpovídá specifikaci

### *2.1 Prototypování*

Z hlediska *prototypování funkcí* potřebujeme popsat, jak jsou vstupní data transformována, přesněji, co je výstupem funkce. Nezajímá nás reprezentace použitých datových struktur ani interpretace operací nad těmito daty, nýbrž výsledek dané funkce. To co od daného nástroje požadujeme by se dalo označit jako *symbolický výpočet*.

Z důvodu konzistence jednotlivých úrovní funkčního modelu je také klíčovým pojmem funkce *abstrakce/konkretizace* popisující (obousměrně) přechod mezi úrovněmi DFD hladin (potažmo "hladinami abstrakce" funkčního modelu)

Prototypování, v souladu s požadavkem na specifikací řízený styl práce, probíhá ve vývojovém prostředí — tj. stále ještě není čas na diskuse o realisaci datových struktur ani algoritmů funkcí z hlediska efektivnosti výpočtu

### *2.2 Tvorba důkazu*

V tomto případě lze na abstraktní specifikaci funkce a její "abstraktní implementaci" (protože stále ve vývojovém prostředí) nahlížet jako na dvě specifikace téhož problému a navíc v rámci jednoho specifikačního nástroje. Za těchto podmínek je možné uvažovat o podpoře tvorby *důkazu správnosti implementace vůči specifikaci*. Důkaz bude spočívat v konstrukci "konverzní mapy", která převede jednu specifikaci na druhou.

K tomu samozřejmě potřebujeme nástroj s vhodnou operační sémantikou, která nám interaktivní vývoj takové konverzní mapy umožní[4].

Pokud jde o vztah "abstraktní" implementace funkce ve vývojovém prostředí, o které je dokázáno, že vyhovuje specifikaci, a "skutečné (efektivní)" implementace, domníváme se, že vztah mezi těmito dvěma specifikacemi je přibližně na stejné úrovni jako konceptuální datový model a jeho implementace pomocí nějaké relační

[4] Myšlenka převodu jedné specifikace (struktury, programu, termu) na jinou je velmi dobře teoreticky propracována v různě rozšířených systémech založených na (typovaném) λ-kalkulu, což je také systém ECC implementovaný v nástroji LEGO

databáze. Totiž, převod bylo by možno udělat "poloautomatickou" cestou za asistence vývojáře nebo kodéra.

Vzhledem ke dvěma různým typům služeb, které od podpory vývoje pomocí formálních metod specifikací očekáváme, totiž prototypování a důkaz správnosti (abstraktní) implementace, navrhujeme použít pro tyto účely dva různé nástroje. Systém OBJ3 je založen na teorii algebraických specifikací se jeví jako vhodný nástroj pro prototypování funkcí, zatímco systém LEGO poskytuje vhodnou operační sémantiku pro vývoj důkazu (resp. tvorbu konverzní mapy).

Oba dva nástroje nyní krátce představíme (větší pozornost budeme věnovat systému LEGO, protože přináší značně odlišný způsob uvažování do programátorského světa). Oba nástroje budou předvedeny na malém modelovém příkladu.

## 3. OBJ3 a LEGO

V posledních 15 letech je v teorii programování a softwarovém inženýrství algebraickým metodám specifikací věnováno stále více pozornosti. Narozdíl od jiných metod a postupů vychází tyto "zespoda" — od velmi dobře formulovaných a konzistentních teoretických základů v matematice až k (účelově omezeným) implementacím — nástrojům pro formální specifikace[5].

### *3.1 OBJ3*

OBJ3 je přepisovací systém vyvinutý v 80. letech na univerzitě ve Stanfordu. Umožňuje provádění symbolických výpočtů a zajišťuje typovou kontrolu algebraických specifikací. Jeho vyjadřovací jazyk je velmi podobný specifikacím algeber v matematice — každý objekt (algebra) má definovány nosiče (datové typy) a operace nad nimi. Operace se definují nejprve syntakticky (arita operace, typy operandů a typ výsledku) a pak následuje popis sémantiky operace ve formě axiomů (rovností resp. podmíněných rovností). Tyto axiomy se pak používají při symbolickém výpočtu — totiž levá strana pravidla se nahradí pravou. Jazyk OBJ3 není komplikovaný a pro programátora je snadno čitelný.

Ještě dodejme, že OBJ3 umožňuje definovat parametrické (generické) moduly a má ještě další možnosti, které však sahají za rámec tohoto příspěvku. Další podrobnosti lze najít v [9].

### *3.2 LEGO*

LEGO je systém, který se od OBJ3 výrazně liší. Svými autory je klasifikován jako asistent pro tvorbu důkazů (*Proof Assistant*). Vznikl v 90. letech na univerzitě v Edinburgu.

Jeho hlavním úkolem je udržovat kontext. Kontext v systému LEGO je vlastně báze v systému teorie typů vycházející z typovaného λ-kalkulu. Je to množina

[5] Samozřejmě že za cenu velmi nízké výpočetní efektivity. Ta však není vzhledem k jejich použití, rozhodující.

uspořádaných dvojic *proměnná : typ proměnné*. Každé tvrzení je pak dokazováno v nějakém kontextu (eventuelně prázdném). Výraz B|a:A čteme například: *"Program a je implementací specifikace A v kontextu B"*. Smyslem kontextu je umožnit uživateli s tímto kontextem manipulovat "povoleným způsobem". Manipulovat s kontextem znamená obohacovat jej o nová tvrzení vyvozená z původních nebo naopak, vypouštět z něj neužitečné předpoklady. Povolený způsob manipulace s kontextem je vždy dán aktuální typovou teorií.

Typová teorie je především soubor axiomů (odvozovacích pravidel), které lze v rámci teorie použít pro otypování termů (programů) a pravidel, které umožní s termy manipulovat. Typicky očekáváme nejméně pravidla typu λ-abstrakce a λ-aplikace a další, která dávají teorii dostatečnou sílu pro simulaci programů. Celou teorii a příklady lze nalézt v [1,2,8].

LEGO umožňuje pracovat v různých typových teoriích viz [7]. Zde použitá typová teorie ECC (Extended Calculus of Constructions) vznikla v 90. letech rovněž na univerzitě v Edinburgu. Jak napovídá název, je to rozšíření CC (Calculus of Construction), a to o predikativní typová univerza (Type_0, ..., Type_n) a Σ-typ. Podrobnosti viz [5].

Pro účely tohoto příspěvku postačí konstatování, že ECC poskytuje dostatečnou sílu k tomu, aby se v něm dalo programovat (má hotové importovatelné moduly pro základní datové typy Bool, List, Nat, ..., umožňuje rekursi a také parametrické moduly). Základní konstrukcí práce v takovém prostředí je funkce.

Samotná práce v systému LEGO, které implementuje nyní ECC, pak může postupovat po různých cestách:

- Specifikaci funkce můžeme chápat jako typ v typové teorii. "Abstraktní" implementace je potom funkce (program) daného typu. Pro důkaz správnosti implementace stačí otypovat danou funkci (program) typem její specifikace. K otypování termu (funkce) máme v dané typové teorii k dispozici axiomy. Blíže viz [7].
- Jiný přístup je ten, že máme dvě specifikace a chceme ukázat, že jednu lze implementovat pomocí druhé (například chceme implementovat zásobník pomocí pole, viz [6]). V tomto případě za pomocí "Proof Assistenta" vytváříme konverzní mapu (zobrazení) jedné specifikace na druhou a dokazujeme, že v implementační specifikaci platí axiomy specifikace, kterou jsme implementovali pomocí konverzní mapy.
- Ještě jiná možnost použití systému LEGO je definovat specifikaci jako predikát, který musí být splněn, a potom za pomocí "Proof Assistenta" ukázat, že námi vytvořená (abstraktní) implementace tento predikát splňuje.

Poslední přístup ukážeme na příkladu specifikace řazení a její implementace — metody Quick sort.

## 4. PŘÍKLAD - SPECIFIKACE ŘAZENÍ

V této kapitole demonstrujeme práci v obou specifikačních nástrojích — OBJ3 a LEGO na jednoduchém příkladu specifikace řazení. Jako implementaci obecné specifikace řazení jsme zvolili algoritmus Quick sort. Jedním z důvodů volby této implementace je rekurzivní charater tohoto algoritmu, což se velmi elegantně vyjádří v OBJ3, bohužel však méně elegantně v LEGU. Na druhé straně důkaz správnosti implementace v OBJ3[6] nelze vytvářet tak systematicky jako v LEGU, které je k tomu přímo určeno.

Algebraická specifikace řazení může být zadána následovně:

$$\lambda\, f{:}list(nat) \rightarrow list(nat).\ \forall l{:}list(nat).sorted(l,f(l))$$

Tedy: řazení je fukce *f*, která má jako argument seznam přirozených čísel a vrací opět seznam přirozených čísel, který však splňuje predikát *sorted*.

Dále je nutné specifikovat význam predikátu *sorted*. Můžeme jej chápat jako funkci typu: *(list(nat), list(nat)) → bool*. Jeho intuitivní význam je, že platí pokud je druhý argument seřazenou verzí prvního. Následuje obecná specifikace řazeni a algoritmu Quick sort v OBJ3 a v systému LEGO.

### *4.1 Řazení a Quick sort v OBJ3*

```
obj SORTING[X :: POSET] is
        protecting LIST[X] .
        op sorting_ : List -> List .
        op unsorted_ : List -> Bool .
        vars L L' L'' : List .
        vars E E' : Elt .
        cq sorting L = L if unsorted L =/= true .
        cq sorting L E L' E' L'' = sorting L E' L' E L'' if E' < E .
        cq unsorted L = false if (|L| <= 1) .
        cq unsorted L E L' E' L'' = true if E' < E .
endo
obj QUICKSORT [X :: POSET ] is
        protecting LIST[X] .
        op piv-le : Elt List -> List .
        op piv-gt : Elt List -> List .
        op quicksort_ : List -> List .
        vars E E' : Elt .
        var L : List .
        eq piv-le(E, nil) = nil .
        cq piv-le(E, E' L) = piv-le(E, L) if E < E' .
        cq piv-le(E, E' L) = E' piv-le(E, L) if not E < E' .
        eq piv-gt(E, nil) = nil .
```

[6] Provést důkaz tohoto problému v OBJ3 je též možné, nicméně z důvodů prostorových jej zde neuvádíme.

```
        cq piv-gt(E, E' L) = E' piv-gt(E, L) if E < E' .
        cq piv-gt(E, E' L) = piv-gt(E, L) if not E < E' .
        eq quicksort nil = nil .
        eq quicksort E L = (quicksort piv-le(E, L)) E (quicksort piv-gt(E, L)) .
    endo
```

### *4.2 Řazení a Quick sort v systému LEGO*

Před uvedením definice řazení je třeba ještě zavést možnost zpracování rekursivní funkce. Prakticky jde pouze o zavedení Turingova operátoru pevného bodu [1,2] do kontextu funkcí ECC. Následující implementace je převzata z [5]. K vlastní syntaxi zápisu v systému LEGO jen několik poznámek:

- *Type* a *Prop* jsou vnitřní univerza ECC.
- *[base : A -> B]* značí, že funkce *base* je typu *A->B*.
- *[f2 : nat->nat = [x:nat]x]* značí, že funkce *f2* je typu *nat->nat* a dle své definice pro každou vstupní proměnnou *x:nat* vrací *x*.
- *nil_test (tail l)* je aplikace funkce *nil_test* na zbytek seznamu *l* získaný aplikaci funkce *tail* na *l*.
- (* general recursion scheme *) je komentář.

Tedy nejprve zavedení rekurze:

```
(* general recursion scheme *)

[A,B|Type]
[end : A->bool] [base : A->B];
[scheme : A->(A->B)->B] [w : A->nat];
[rec_aux = nat_rec
        base
        ([_:nat] [upton:A->B] [a:A] if (end a) (base a) (scheme a upton))]
[recurse [a:A] = rec_aux (w a) a];

Discharge A;
```

Následuje definice predikátu *sorted*:

```
(* auxillary function *)
[let [A,B|Type] = [a:A] [f:A->B]f a];

(* conversions *)
[f1 : unit->nat = [x:unit]zero];
[f2 : nat->nat = [x:nat]x];
(* termination condition *)
[chk_nxt_cond  [l : list nat] = nil_test (tail l)];

(* base case *)
[chk_nxt_base  = [l:(list nat)]true];

(* recursion scheme *)
```

```
[chk_nxt_scheme [l : list nat] [nxt : (list nat) -> bool] =
   let l
      ([a : list nat]
       if (lt (case f1 f2 (head (tail a))) (case f1 f2 (head a)))
                 false
                 (nxt (tail l)))];

[Chk_nxt_tw =
 recurse chk_nxt_cond chk_nxt_base chk_nxt_scheme (length|?)];

(* order check *)
[Chk_nxt = [l:list nat]Chk_nxt_tw l];

(* predicate sorted *)
[Sorted  : (list nat) -> (list nat) -> Prop =
 [a,b : list nat]
 and (Perm a b) (* result has to be a permutation of input *)
    (is_true (orelse (nil_test (tail b))
                (andalso (le (case f1 f2 (head b))
                             (case f1 f2 (head (tail b))))
                         (Chk_nxt (tail b)))))];
```

A konečně definice funkce *Quick_srt* s pomocnými funkcemi *Piv_gt* a *Piv_le*:

```
(* Piv_gt used by Quick_srt specification *)
[piv_gt_cond [x:nat][l:list nat] = nil_test l];
[piv_gt_base = [x:nat][l:list nat](nil nat)];
[piv_gt_scheme [x:nat][l:list nat] [pv_gt:(list nat)->(list nat)] =
        let l
                ([m:list nat]
                (if (le x (case f1 f2 (head m)))
                        (append (singleton (case f1 f2 (head m)))
                                (pv_gt (tail m)))
                        (pv_gt (tail m))))];

[Piv_gt_tw [x:nat] =
       recurse (piv_gt_cond x) (piv_gt_base x)
 (piv_gt_scheme x) (length|?)];

[Piv_gt = [x:nat][l:list nat]Piv_gt_tw x l];

(* Piv_le used by Quick_srt specification *)
[piv_le_cond [x:nat]][l:list nat] = nil_test l];
[piv_le_base = [x:nat][l:list nat](nil nat)];
[piv_le_scheme [x:nat][l:list nat] [pv_le:(list nat)->(list nat)] =
        let l
                ([m:list nat]
                (if (le x (case f1 f2 (head m))))
                        (pv_le (tail m))
```

```
                    (append (singleton (case f1 f2 (head m)))
                        (pv_le (tail m))))))];

[Piv_le_tw [x:nat] =
        recurse (piv_le_cond x) (piv_le_base x)
 (piv_le_scheme x) (length|?)];

[Piv_le = [x:nat][l:list nat]Piv_le_tw x l];

(* Quicksort specification *)
[quick_srt_cond [l:list nat] = nil_test l];
[quick_srt_base = [l:list nat]nil];
[quick_srt_scheme [m:list nat] [quick_srt:(list nat)->(list nat)] =
        let m
        ([l:list nat]
        (append (append (quick_srt (Piv_le (case f1 f2 (head l)) (tail l)))
                (singleton (case f1 f2 (head l))))
                        (quick_srt (Piv_gt (case f1 f2 (head l)) (tail l)))))))];

[Quick_srt_tw =
        recurse quick_srt_cond quick_srt_base
        quick_srt_scheme (length|?)];

[Quick_srt = [l:list nat]Quick_srt_tw l];
```

### *4.3 Ukázka vývoje důkazu v systému LEGO*

Vývoj důkazu v systému LEGO začíná specifikací cíle.

```
Lego> Goal {l:list nat}Sorted l (Quick_srt l);
Goal
        ?0 : {l:list nat}Sorted l (Quick_srt l)
```

LEGO přijme cíl důkazu a označí jej symbolickou proměnnou *?0*. Zavedeme proměnnou *l* (Intros l) a LEGO jí pomocí dedukce přiřadí typ *list(nat)*. Otypovaná proměnná *l* obohatí kontext. Přitom je též starý cíl (*?0*) nahrazen novým (*?1*).

Pokračujeme expanzí predikátu *sorted* (Expand sorted) a dále provedením *"and introduction"* (andI). Původní cíl *?1* se rozpadne na dva podcíle *?4* a *?5*.

```
Lego> Intros l;
Intros (1) l
        l : list nat
        ?1 : Sorted l (Quick_srt l)
Lego> Expand Sorted;
Expand Sorted
        ?1 : and (Perm l (Quick_srt l))
                (is_true (orelse (nil_test (tail (Quick_srt l)))
                (andalso (le (case f1 f2 (head (Quick_srt l)))
```

```
            (case f1 f2 (head (tail (Quick_srt l))))) (Chk_nxt (tail (Quick_srt l))))))
Lego> andI,
and Intro
        l : list nat
        ?4  Perm l (Quick_srt l)
        ?5 : is_true (orelse (nil_test (tail (Quick_srt l)))
                (andalso (le (case f1 f2 (head (Quick_srt l)))
                (case f1 f2 (head (tail (Quick_srt l))))) (Chk_nxt (tail (Quick_srt l)))))
Lego> Next 4; Expand Quick_srt;
        ...
```

Dále bychom pokračovali indukcí podle délky *l* v obou podcílech *?4* a *?5*. Poznamenejme ještě, že LEGO přirozeně podporuje rekursi na datových typech díky jejich rekursivnímu zavedení (viz [7]).

## 5. ZÁVĚR

Konstatovali jsme jisté nedostatky v současných CASE systémech z pohledu přechodu na specifikací řízený styl práce. Zaměřili jsme se pouze na funkční model systému. Například v datovém modelu je způsob práce se specifikací (a *tím* také jistá paralelní tvorba dokumentace a ostatní výhody specifikací řízeného stylu práce) uspokojivě řešen používáním konceptuálních modelů dat. Jako možné řešení jsme představili dva různé nástroje pro práci se specifikací funkcí a velmi stručně demonstrovali, jakým stylem pracují.

Tento příspěvek je spíše než řešením nedostatků funkčního modelu současných CASE systémů náznakem jedné z možných cest řešení. Je to spíše výchozí bod pro další práci.

Dílčí problémy, které je nutno vyřešit, jsou například: způsob integrace nástrojů pro algebraické specifikace a teorii typů s CASE systémy, vytvoření vhodných *"primitiv"* pro specifikaci a vývoj důkazů v softwarovém inženýrství nebo automatický či poloautomatický přechod od *"abstraktní"* implementace k *"efektivní"* implementaci či přechod mezi nástrojem pro prototypování funkcí a asistentem pro důkaz.

## 6. LITERATURA


[1] Barendregt, H.,P.: *Functional Programming and Lambda Calculus*. Handbook of Theoretical Computer Science. Editor: J. van Leeuwen Elsevier Science Publishers B.V. 1990, str. 323—360.

[2] Barendregt, H.,P.: *Lambda Calculi With Types*. Handbook of Logic in Computer Science. Editor: S. Abramski, D.M. Gabbay, T.S.E. Maibaum. Oxford University Press, New York, 1992, str. 118—279.

[3] Hoffman, M.: *Formal Development of Functional Programs in Type Theory - A CASE Study*. Dept. of Computer Science, University of Edinburgh 1992.

[4] Girard,J.Y. - Lafont,Y. - Taylor,P.: *Proofs and Types*. Cambridge University Press, 1989.

[5] Luo, Z.: *ECC An Extended Calculus of Construction* Dept. of Computer Science, University of Edinburgh 1991.

[6] Luo, Z.: *Program Specification and Data Refinement in Type Theory*. Dept. of Computer Science, University of Edinburgh 1991.
[7] Luo,Z. - Pollack,R.: *LEGO Proof Development System: User's Manual*. Dept. of Computer Science, University of Edinburgh, 1992.
[8] Nordstrom, B., Petersson, K., Smith, J.M.: *Pogramming in Martin Lof's Type Theory*. Oxford Science Publications, Oxford 1990.
[9] Goguen,J.A. - Wingler,T. - Meseguer,J. - Futatsugi,K. - Joanound,J.P: *Introducing OBJ3*. SRI Computer Science Laboratory Technical Report, SRI-CSL-92-03, Menlo Park 1992.