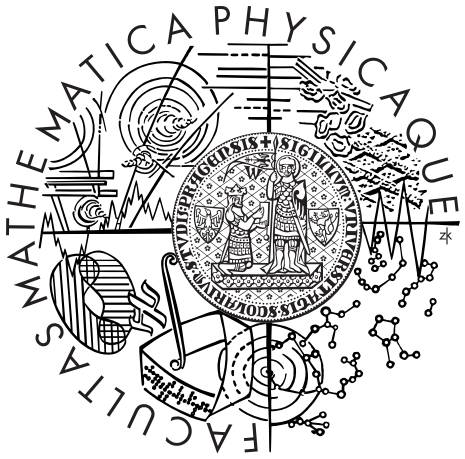
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Miroslav Kratochvíl

Nízkoúrovňový funkcionální programovací jazyk

Katedra softwarového inženýrství

Supervisor of the master thesis: RNDr. David Bednárek, Ph.D.

Study programme: Informatika

Specialization: Teoretická informatika

Prague 2015

Dedicated to people who emit machine code directly.

I would like to thank everyone who took part in creating this thesis, especially to the supervisor, RNDr. David Bednárek, Ph.D., who provided lots of practical advice that steered the thesis into this form, and RNDr. Jan Hric and Bc. Vít Šefl, who have both inspired and helped me to several solutions of problems with type systems and functional programming.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Nízkoúrovňový funkcionální programovací jazyk

Autor: Miroslav Kratochvíl

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek, Ph.D.

Abstrakt: Cílem práce je prozkoumat možnosti implementace kompilátoru nízkoúrovňového funkcionálního jazyka. Předpokládá se zhodnocení teoretických vlastností funkcionálních jazyků, možných omezení vyplývajících z absence běhové podpory kódu a implementace kompilátoru jazyka, který demonstruje některé vybrané vlastnosti.

Klíčová slova: návrh programovacích jazyků, funkcionální programování, překladače, typovaný lambda kalkulus

Title: Low-level functional programming language

Author: Miroslav Kratochvíl

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D.

Abstract: The goal of this thesis is to explore the current possibilities of implementation of compilers of low-level functional languages. It is expected to evaluate theoretical possibilities of functional programming languages, possible limitations that arise from the absence of run-time code support in low-level environment, and to implement a language compiler that demonstrates some chosen properties.

Keywords: programming language design, functional programming, compilers, typed lambda calculus

Contents

Introduction

Compilers of programming languages, used as tools for streamlining and optimizing the communication of the programmer with the machine, have been extensively developed since the half of 20^{th} century. There are various programming languages optimized for solving different tasks on different platforms, different efficiency constrains on time, space, expression format or required programming proficiency.

With some generalization, currently available compilers may be sorted into several overlapping categories by the possibilities the programming language exposes for the programmer and the properties of resulting compiled code form. For our reasoning we will consider following three informal measures:

- First, what level of run-time support does the resulting code (or, usually, the compiled program) need. This marks the languages as being interpreted, byte-code-interpreted (or running in a VM) or translated into machine code with or without supporting runtime. Compare respectively languages like Scheme, Java, OCaml and C.

  This *runtime size* is important for considerations about performance of the language – whether it will benefit from either garbage collection or exact memory allocation, from simple code execution by interpreter or from compilation and heavy optimization for specific target.

- Second, what level of *expressiveness* does the programming language provide. Consider, for example, a relatively simple idea of non-deterministic solution of some puzzle implemented in PHP in contrast to Scheme. While both languages are similar in terms of being interpreted and having no serious type system, scheme program can use the language features to construct straightforward, Prolog-like backtracking using the `(amb)` operator [Sit], which handles the situation in a very elegant, efficient and readable way, while the PHP program is forced to simulate the non-determinism, backtracking and decision points "by hand", leaving the programmer with a lot of non-semantic implementation-specific code that is usually hard to maintain or reuse.

  For another view on expressiveness, compare C and C++ – while the language concept is extremely similar (both have been anecdotally entitled and used as "portable assemblers" [1]), the programming

[1] in the case of C++, as a "portable assembler with a really exaggerated macro system"

paradigms added to C++ make it very easy to construct highly complex programs (notably using the STL library), but some properties needed to support the paradigms, for example the dependency of most of the standard library on hidden implicit allocations, C++ name decoration or the method of virtual function call prohibit those features from being efficiently used in development of specific operating system kernels and similar tools.[2]

- Third, to what level is the language designed to be easily understood and manipulated by tools that process it, most notably by compilers, optimizers and verification tools (including humans). *Apprehensive* language should have a structure that does not force the programmers to write a code that would be hard to comprehend, prevents code maintenance difficulties or errors resulting from code complexity; and similarly, it should have very simple representation and internal structure that would make it easy to process by other programs.

  Functional languages are excellent from this point of view, due to the omnipresent locality principle and controllable side effects of constructions. Logic-based programs are easily verified for constraints on their results, as the program sources are basically just lists of the same applied constraints. Imperative languages do not deny such principles, but their compilers are usually forced to expect a generalized non-ideal case where many easy optimizations may not be readily visible or available.

Motivation for this thesis is very simple: The separations above hint that a generally-applicable language that combines all three properties could be very useful; but, to the best of author's knowledge, such programming language currently does not exist and it is therefore necessary to create one.

Approach to the solution chosen for this thesis is to try to proceed with the slow, successive shift of the paradigms that can be observed on current programming languages. In last ten years, several low-level programming languages (most notably C++, with the introduction of the C++11 standard) have been extended to also allow many programming primitives and styles from the functional and declarative programming.

At the same time, programming languages that are functional, logical or meant as interpreted by design are getting various extensions that allow them to produce high-performance code, by using some kind of language restriction, new compiler or external conversion tool. Examples of this

[2]One of the most famous opinions related to similar topic may be found in [Tor].

movement include the HHVM compiler for PHP [hhv], the PyPy project [pyp], various languages with small runtime applicable for highly-performing solutions of specific tasks (which include current JavaScript engines, see also the Related research section for more), or, importantly, various low-level extensions for functional languages, especially the practical improvements of Haskell compilers and construction of several kernels based only on functional languages.

This thesis aims to follow this development by producing a combination of the paradigms that would fill one of the empty spaces on the language list, by combining the functional paradigms and syntax from Haskell with the execution and memory management model of C-like languages. The result is expected to carry both the good properties of Haskell (for the expressiveness and apprehensivity) and those of C and C++ (almost no run-time requirements, and the ability to construct all the library code directly in the language[3]).

Combining those paradigms is not a very straightforward process. Functional programming usually requires some run-time support in form of the automated memory manager and garbage collector. The aim of this thesis is to identify exact language primitives that require garbage collection, and either to provide a solution that runs without garbage collector, or to alter the language so that the primitive is replaced by a less-demanding one, effectively removing the whole need for automated memory management. The main goal is then to provide an implementation of a compiler of a subset of the planned language that, as a proof-of-concept, demonstrates the viability of those results.

We can observe many specific cases where resulting programming language could significantly improve software development. Usual points where the needs for easy apprehension, expressiveness and performance meet are following:

- Common operating system kernels have to perform extremely well, even in constrained situations or embedded hardware. Common implementations in C-like languages that satisfy this property bring lots of issues with code stability and security, also making any verification hard. Examples of such situations can be seen in seL4 movement (especially their tools for interface definitions) or in traditional hunting of security exploits in Linux kernel code.

- Scientific tools must be expressive and apprehensive to allow scientists without formal education in programming run research-oriented

[3] Just like in the case of STL from C++.

calculations, but implementations usually suffer from not being optimized for running complex programs. Consider MATLAB, that, on the one hand, implements very efficient routines (for example for handling large matrix-structured data), but the language itself is partially interpreted, possesses no efficient data structures and is hard to connect with other tools because of the limited input/output possibilities.

- Cryptography and related software requires a lot of programming from scientists that do not have background in how non-mathematical or non-cryptography attacks work, almost directly resulting for example into problems like OpenSSL buffer-overflow bugs (see [LLH+10] for another systematical approach that patches the same problem).

  Having an expressive language that can provide a better separation of software-related and cryptography-related concepts, but is able to combine them into a highly performing package, could ease the cooperation on security-related code and prevent human errors.

- Similar situations arise also with needs for reliable or highly available application-specific software where the language runtime must fit and perform optimally on a restricted or embedded hardware, but the used language should be apprehensive enough to allow easy formal verification (that is usually required).

  Examples include automated systems for driving various dangerous vehicles, machines, or collectively environments where errors cost money or lives.

Related research

There has already been much research effort done on existing programming languages that covers great amount of language features described here.

A lot of work was done on theoretical features of functional languages, especially on various combinations of type systems and their features. Related literature and results are referenced directly from the thesis.

Big amount of research with similar target is done in Haskell community, producing low-level Haskell tools. Such efforts include Haskell kernels (notably hOp, House or LightHouse, see for example [HJLT05]), research on memory safety (notably the dissertation thesis of Rebekah Leslie [Les11] covering the gap from safe to unsafe memory by simple operation isolation)

and functional generators of safe non-functional code (for example the Atom domain-specific language).

Habit language [HAS10] is a project of Portland State university HASP group, that states the need for similar language, although choosing a little different feature set than this thesis. Most recent publications from the project discuss the language features and provide a clear specification for implementation. To the best of author's knowledge, no implementation of Habit is available yet.

Many "mini-languages" that target similar environments have also emerged in last decades. These include Rust, clean, Nim language, and lots of related efforts.[4] Such languages usually bring interesting new concepts and views, but almost all require heavy runtime support.

Thesis layout

First chapter of the thesis is partly a crash-course to the lambda calculus intended for a reader with no background in actual functional programming, and partly a quick overview of several type systems that are usually not a part of lambda-calculus courses. We revisit some basic definitions and results about the calculus, especially the definitions that will be used later for the description of the new language, provide a reader with some insight about the reasons of undecidability of several associated problems, and explain brief details about the common type systems.

Details about inner workings of compilers of functional languages are presented in the second chapter, including complete algorithms and discussion about related difficulties. The chapter is intended as a complete overview of the compiling process, connecting the whole chain of operations from the input program to resulting generated code.

Approach to the construction of the low-level functional language, along with details concerning the implementation of proof-of-concept compiler that processes the simple functional language into low-level code can be found in the third chapter.

Appendix of the thesis contains a brief tutorial to usage and modification of the compiler.

[4] Not very long ago, even the widely popular Ruby was a mini-language.

1. Lambda calculus

This chapter is intended as a quick, condensed and partly incomplete course to the basics of lambda calculus, serving as a reference for readers not experienced in the topic. The rest of the readers may skip to the second half of the chapter, starting by section 1.3 that describes several type systems and concepts that were developed for practical programming.

Lambda calculus was introduced by Church [Chu85] as a tool for describing foundations of logic and mathematics, especially for further generalization of the concept of computable functions as seen in computability theory.

Simplicity of the syntax and several strong properties of the system make it an excellent choice for programming language modeling. Although there is little to be gained from actual lambda calculus implementation, there are many interesting properties easily proven on the original simple system that, as shown later in this thesis, persist even when enriching the system with various primitives to enable practical usage.

The calculus itself is a term-rewriting system that works with abstractions and applications (similar to function body and function call from common programming systems). Abstraction, as the creation of simple anonymous function with parameter is denoted by $\lambda x.Y$, and application, written $A\ B$ is a "call" of this function that replaces all occurrences of the abstracted parameter in the abstraction A by supplied term B.

Example computations on lambda calculus enriched by a common set of integer constants, symbols and equalities, together with its specific reduction $\rightarrow_{\lambda_i}$ look very intuitive:

$$(\lambda x.x)0 \rightarrow_{\lambda_i} 0$$

$$(\lambda y.(\lambda x.y + (x \cdot 2))\ 5\ 3 \rightarrow_{\lambda_i} (\lambda x.5 + (x \cdot 2))\ 3 \rightarrow_{\lambda_i} 5 + (3 \cdot 2) \rightarrow_{\lambda_i} 11$$

In this section, we formally define the lambda calculus, show several important properties that are possible to prove, describe typed lambda calculi and several techniques that aid using lambda calculus in common programming environments.

1.1 Lambda calculus without types

1.1.1 Formal definition

λ-terms

Definition 1.1 (λ-term)**.** *Given a set of variables x_i, λ-terms are defined recursively:*

- *Every variable x_i is a λ-term,*
- *if t is λ-term, then $(\lambda x_i.t)$ is λ-term, called λ-abstraction,*
- *if t, s are λ-terms, then $(t\ s)$ is λ-term called λ-application.*

Definition is usually shortened to a very simple formal grammar, for a language Λ of λ-terms:

$$V ::= v_0 | v_1 | v_2 | \cdots$$
$$\Lambda ::= V | (\lambda V.\Lambda) | (\Lambda\ \Lambda)$$

For brevity, all terms mentioned from this in this thesis will be implicitly considered λ-terms, if not specified otherwise. Also, following shortcuts and associativity rules are commonly used:

$$\lambda xy.A = \lambda x.(\lambda y.A)$$
$$ABC = (A\ B)\ C$$

Substitution

Free variable is intuitively a variable that is not bound by any abstraction.

Definition 1.2 (Free variables)**.** *Set of free variables* $\text{free}(t)$ *of term t is defined recursively on the structure of the term:*

- *If t is a variable v_i, then* $\text{free}(t) = \{v_i\}$,
- *for application,* $t = A\ B \implies \text{free}(t) = \text{free}(A) \cup \text{free}(B)$,
- *for abstraction,* $t = (\lambda v_i.A) \implies \text{free}(t) = \text{free}(A) \setminus \{v_i\}$.

If a variable belongs to $\text{free}(t)$*, it is called* free in t.

Substitution is an operation that replaces all occurrences of a variable in the term by some other term. Some care must be taken not to overwrite or produce bound variables, as such substitution might change the semantics of the term.

Definition 1.3 (Substitution)**.** *If variable x does not occur free in term S, the result of substituting the variable x by term S in the term T is denoted as $T[x := S]$ and is defined inductively on the structure of T as follows:*

- $x[x := S] = S$,
- $y[x := S] = y$ *(where x and y are different variables),*
- $(\lambda x.A)[x := S] = (\lambda x.A)$ *(because x is bound in A),*
- $(\lambda y.A)[x := S] = (\lambda y.A[x := S])$ *if x and y are different and y is not free in S,*
- $(A\ B)[x := S] = A[x := S]\ B[x := S]$.

In the fourth case, when substitution would fail because of free-variable check, a 'variable conversion' can be employed: One can find some variable v_i that is not free in both A and S, and continue the substitution on $(\lambda v_i.A[y := v_i][x := S])$.

Similar concept is usually called α-conversion,[1] basically stating that $\lambda x.A = \lambda y.A[x := y]$. Most current systems simply avoid name collision and (costly) free-variable checking and conversion by assigning unique names to all captured variables before anything is done with the lambda program by running one "global" α-conversion.

Theorem 1.1 (Substitution lemma)**.** *Order of the substitutions is not important, specifically*

$$(\forall A, B, C)A[x := B][y := C] = A[y := C][x := B[y := C]]$$

Proof. By induction on the structure of term A. □

[1] α-conversion was originally included as a first rule in λ-calculus evaluation ruleset. Today it is usually omitted in most descriptions, leaving a visible remain in the fact that β-reduction is named only after the second letter of the alphabet.

β-conversion

β-conversion is the basic evaluation rule of lambda calculus, giving semantic meaning to both applications and abstractions.

Axiom 1.1 (β-conversion)**.**

$$(\lambda x.A)\ B = A[x := B]$$

For completeness and brevity of reasoning that will follow, the usual set of equality axioms can be easily extended with following rules:

$$\begin{aligned} M = N &\implies N = M \\ &\implies MZ = NZ \\ &\implies ZM = ZN \\ &\implies \lambda x.M = \lambda x.N \end{aligned}$$

η-conversion is not important for actual computation, but can vastly simplify situations in functional language optimization. [2]

Axiom 1.2 (η-conversion)**.**

$$(\lambda x.Ax) = A$$

Traditionally, the terms that can be used as functions are called combinators. Commonly used combinators include the identity $I = \lambda x.x$, selection $K = \lambda xy.x$ or substitution $S = \lambda xyz.xz(yz)$.[3] Very-well-known Y combinator serves as a rather useful way to work with recursive functions:

Theorem 1.2 (Fixed-point)**.** *For each λ-combinator F there exists a fixed point X so that $X = FX$.* [4]

Proof. Take $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$ and set $X = YF$.

Then

$$\begin{aligned} X = YF &= (\lambda x.F(xx))(\lambda x.F(xx)) \\ &= F((\lambda x.F(xx))(\lambda x.F(xx))) \\ &= F(YF) = FX \end{aligned}$$

□

[2] Constructions that do not require η-conversion are usually called "point-free" [CP04].

[3] SKI calculus created only from the mentioned three combinators is Turing-complete.

[4] The fixed-point works "the other way" than the fixed points known from real analysis. FX does actually not reduce to X (at least not in general case), but the reduction of X generates the bodies of given F.

1.1.2 T-completeness

In this section we will show that lambda calculus can be used to compute any recursively enumerable function. We will use a reduction to traditional arithmetic function system.

Because of the nature of Church numerals defined below, we will use following shortcut for multiple applications on terms A and B:

$$A^0 B = B$$
$$A^i B = A^{i-1}(AB)$$

Church numerals

Consider a set of constants $\mathbf{c}_0, \mathbf{c}_1, \ldots$ that are defined as follows:

$$\mathbf{c}_n = \lambda f x.f^n x$$

Theorem 1.3 (J. B. Rosser)**.** *There are arithmetic combinators A_+, A_*, A_e such that*

$$A_+ \mathbf{c}_a \mathbf{c}_b = \mathbf{c}_{a+b}$$
$$A_* \mathbf{c}_a \mathbf{c}_b = \mathbf{c}_{ab}$$
$$b \neq 0 \implies A_e \mathbf{c}_a \mathbf{c}_b = \mathbf{c}_{a^b}$$

Proof. Reference to the detailed proof can be obtained in [BDS13, Proposition 2.2.2].

First, observe that

1. $(\mathbf{c}_n x)^m y = x^{nm} y$, by induction on m, for $m = 0$ the equality holds as $y = y$. Then, for $m + 1$, we can derive $(\mathbf{c}_n x)^{m+1} y = \mathbf{c}_n (\mathbf{c}_n x)^m y = \mathbf{c}_n x (x^{mn} y) = x^n x^{mn} y = x^{(m+1)n} y$.

2. $(\mathbf{c}_n)^m x = \mathbf{c}_{n^m} x$, by the exactly same induction with the result rewritten to Church form.

We can now define the combinators as

$$A_+ = \lambda xypq.xp(ypq)$$
$$A_* = \lambda xyz.x(yz)$$
$$A_e = \lambda xy.yx$$

A_+ works by induction on b parameter. A_* works correctly exactly by using the equation 1.

Using the equation 2, we can show that $A_e \mathbf{c}_a \mathbf{c}_b = \mathbf{c}_b \mathbf{c}_a = \lambda x.\mathbf{c}_n^m x = \lambda x.\mathbf{c}_{n^m} x$, which is equal to $\mathbf{c}_{n^m}$ by η-conversion. □

Recursive functions

Partial recursive functions are usually defined as mappings $\mathbf{N}^n \to \mathbf{N}$ and constructed from the six basic operations (U_n^i, S, Z, composition, primitive recursion and minimization). This system of arithmetically recursive function can be used to define any Turing-computable function. Following proof was first shown in [K$^+$36].

We will show how it is possible to λ-define any such function by simple rewriting of their definitions. In the rewriting process, we will substitute any natural numbers used by their corresponding Church numerals and construct the (informally specified) homomorphism from the algebra made of natural numbers and arithmetic functions to the one made of corresponding numerals and λ-terms.

Initial functions are defined and λ-defined as follows:

$$Z(n) = 0 \to Z = \lambda x.\mathbf{c}_0$$
$$S(n) = n + 1 \to S = A_+\mathbf{c}_1$$
$$U_k^i(n_1, \ldots, n_k) = n_i \to U_k^i = \lambda x_1 x_2 \ldots x_k.x_i$$

Composition of functions g and $h_1, \ldots, h_k$, if they are already λ-defined as G and $H_1, \ldots, H_k$, is rewritten[5] as

$$f(\vec{n}) = g(h_1(\vec{n}), h_2(\vec{n}), \ldots, h_k(\vec{n})) \to F = \lambda\vec{x}.G(H_1\vec{x})(H_2\vec{x})\ldots(H_k\vec{x})$$

Primitive recursion is defined as a "for loop" with supporting functions g and h:

$$f(0, \vec{n}) = g(\vec{n})$$
$$f(k + 1, \vec{n}) = h(f(k, \vec{n}), \vec{n})$$

For the purpose of following needed functional programming we define several simple control structures:

- Boolean types as **true** $= K$ and **false** $= K_*$[6].
- `if` construct - for some Boolean B, we will write **if** B **then** T **else** F as BTF.

[5] Sequences like $x_1, \ldots, x_l$ are shortened to vector $\vec{x}$ as usual.

[6] $K_* = \lambda xy.y$

- Pair $[A, B]$ is constructed as $\lambda x.xAB$, so that $[A, B]\mathbf{true} = A$ and $[A, B]\mathbf{false} = B$.

- Zero-check is constructed as $\mathbf{zero} = \lambda x.x(\mathbf{true}\ \mathbf{false})\mathbf{true}$. Obviously, $\mathbf{zero}\ \mathbf{c}_0 = \mathbf{false}$ and $(\forall i)\mathbf{zero}\ \mathbf{c}_{i+1} = \mathbf{true}$.

Primitive recursive function f is then λ-defined from already defined G and H in several steps.[7]

- First, we create a combinator T that computes one iteration of the cycle, thus produces a pair $k+1, f(k+1)$ from $k, f(k)$. Take

$$T = \lambda a.[S(a\ \mathbf{true}), H(a\ \mathbf{false})(a\ \mathbf{true})]$$

.

If we "call" this T on a pair $[\mathbf{c}_k, \mathbf{c}_{f(k)}]$, it visibly increments the first numeral, and correctly calls H on the second numeral.

- We take T and copy it k times by numeral iteration, thus receiving

$$[\mathbf{c}_k, F\mathbf{c}_k] = T^k[\mathbf{c}_0, \mathbf{c}_{f(0)}] = \mathbf{c}_k T[\mathbf{c}_0, \mathbf{c}_{f(0)}]$$

- After adding G instead of $F\mathbf{c}_0$, the result is picked from the pair by **false**, and the iterator for k is abstracted to get the final λ-definition of f:

$$F = \lambda k.kT[\mathbf{c}_0, G]\mathbf{false}$$

Minimization, needed for definition of all partial recursive functions, cannot be defined by simple iteration – possible infinite iteration would require the impossible infinite term. Instead, we employ the fixpoint combinator Y in following lemma to produce and correctly use an endless stream of function bodies.

Lemma 1.1 (Context lemma)**.** *For each term C with free variables f and x*

$$(\exists F)(\forall X)FX = C[f := F][x := X]$$

[7]Note that parameters $\vec{n}$ of G and H were omitted for brevity, but as they are constant through whole computation, adding them as additional parameters to the result is done simply by prepending $\lambda\vec{n}$.

Proof. We set F as follows, and continue to evaluate the substituted term to get C in the correct form.

$$\begin{aligned} & F = Y(\lambda f x.C) \\ \implies & F = (\lambda f x.C)F && (= (\lambda f x.C)Y(\lambda f x.C)) \\ \implies & F = \lambda x.C[f := F] \\ \implies & Fv = C[f := F][x := v] \\ \implies & (\forall X)FX = C[f := F][x := X] \end{aligned}$$

□

Although it looks like a simple substitution, context lemma gives a straightforward way to circumvent the basic limitation of substitution – here, the "substituted" value of f can contain free occurrences of f itself. Producing a fully substituted term is impossible in that case (since it would have to be infinite), but producing a term that would behave equally under β-reduction is possible.

That gives a straightforward way to create a recursive (in terms of self-reference) function by taking the body of the function and replacing all its self-referencing variables with the same function body. Note that context lemma works flawlessly also for multi-parameter function, as x and X in the lemma above can be extended to $\vec{x}$ and $\vec{X}$ without any impact on the proof.

For the case of recursive minimization $f(\vec{n}) = \mu_x\{g(x, \vec{n}) = 0\}$, we can easily λ-define the function using the tools described earlier:

- With context lemma, we can construct recursive H such that

  $$H = \lambda x \vec{n}.(\mathbf{zero}\ Gx\vec{n})\ x\ (H(Sx)\vec{n})$$

  Note the if-then-else construction using the Boolean type.

- Then f is defined by

  $$F = \lambda \vec{n}.H\mathbf{c}_0\vec{n}$$

Theorem 1.4. *Lambda calculus is Turing-complete.*

Proof. Any function from the Turing-complete arithmetic recursion system can be inductively translated to λ-definable terms as shown above and then evaluated in λ-calculus. □

Besides the proof for the computation power of lambda calculus, the completeness theorem gives an interesting insight: Using the fact that the converse statement – that the λ-terms can be enumerated by Turing machine – also holds, we easily show that the notions 'recursively computable', 'λ-definable' and 'Turing-computable' are equivalent.

1.1.3 Halting

Obviously, the ability to simulate a Turing machine implies that determining whether evaluation of given λ-term will halt is an undecidable problem. One of the most useful undecidability results is that of Church, proving that λ-calculus is also an undecidable theory:

Theorem 1.5 (Church)**.** *For any λ-term X, the set*

$$\{M | M =_\lambda \mathbf{true}\}$$

is not recursive.

Proof. See (LINK church). For our purposes, almost identical result can be obtained from the set $\{M | M =_\lambda \mathbf{c}_0 =_\lambda \mathbf{false}\}$ which is equivalent to the non-recursive arithmetical set of all programs that evaluate to zero. □

For an easy demonstration of halting properties of "λ-programs", compare following terms:

- $(\lambda x.x)$ can not be evaluated by any conversion rule, so we assume that it is the result of the evaluation that halted.
- $\Omega = (\lambda x.xx)(\lambda x.xx)$ has only one possible applicable evaluation (β-conversion of the application), which leads back to Ω and repeated evaluation will never halt.
- $(\lambda x.xxx)(\lambda x.xxx)$ will not halt, and the evaluation will never come to the same state twice.
- With $P = [\lambda x.x, \Omega]\mathbf{true}$ the situation gets more complicated. This program *may halt* – if the evaluation engine chooses the correct application, the result is $(\lambda x.x)$. If the evaluation engine always tried the application in Ω, it would never halt.

Normalization

Formal definition of what exactly is a "result" of computation is traditionally given by β-normal forms.

Definition 1.4 (β-nf)**.** *λ-term in the form $(\lambda x.A)\ B$ is called a* β-redex.
λ-term that doesn't contain a β-redex is called to be a β-normal form.

Definition 1.5 (β-reduction)**.** *Binary relation $\rightarrow_\beta$ (called "β-reduces in one step to") is defined recursively as*

- $(\lambda x.A)B \to_\beta B[x := A]$
- $M \to_\beta N \implies (ZM \to_\beta ZN) \wedge (MZ \to_\beta NZ) \wedge ((\lambda x.M) \to_\beta (\lambda x.N))$.

$\twoheadrightarrow_\beta$ ("β-reduces to") is defined as a transitive and reflexive closure of $\to_\beta$.
$=_\beta$ ("is β-convertible to") is defined as a transitive and symmetric closure of $\twoheadrightarrow_\beta$.

For example,

- $(\lambda x.x)y$ has β-nf y,
- $(\lambda x.x)y =_\beta (\lambda x.y)z$.

Definition 1.6. *Term X is called* normalizing *if there exists some β-nf Y so that $X \twoheadrightarrow_\beta Y$. This is usually called "X has a normal form".*
Term is strongly normalizing *if all possible reduction paths lead to β-nf.*

For example, the aforementioned term $[(\lambda x.x), \Omega]\mathbf{true}$ is normalizing, and $I(xI(xI(xIx)))$ is strongly normalizing, because no matter in which order the I's are reduced, there is always a β-nf result $xxxx$.

Conditions necessary for the term to be (strongly) normalizing are best stated by λ-type systems later in this thesis. For simply normalizing terms, following theorem due to Curry gives a *reduction strategy* that finds the β-nf.

Theorem 1.6 (Leftmost normalization strategy)**.** *If A is normalizing, it can be evaluated to β-nf by iterated reduction of its leftmost redex.*

Proof. Formal proof is given in [BDS13, Section 13.2.2].

Informally, each leftmost redex must be reduced at some point (without the reduction, it would never disappear because it can not be discarded by any other redex on the left) and, by substitution lemma, there is no point in postponing its reduction after the reduction of any non-leftmost redexes, because it will never aid the possibility of halting. □

Given strategy is usually called *lazy evaluation*. Note that although many functional programming languages claim to use lazy evaluation, in most cases the extent of applied laziness is limited by practical considerations – in most cases, lazy evaluation leads to a lot of substitutions and large temporary structures. Other evaluation methods (for example the "leftmost primitive first" from Scheme) obey locality principles and do not put such stress on the term-rewriting engine.

1.2 Basic type systems

Informally, type systems are collections of rules used to assign and prove properties on the program code, in our case on λ-terms. The properties assigned may vary by the type system used, but they usually include

- making a decision that the program is without (certain kinds of) errors and will not fail at runtime
- checking that the program will terminate by strong normalization theorems,
- assigning some semantic meaning represented by a type to a program, and machine-checking if the program can have the expected type
- formalization of theorem proving methods.[8]

1.2.1 Simply typed lambda calculus

Types in lambda calculus are usually assigned by one of two syntactical systems, *Curry* and *Church* typing.

Definition 1.7 (Typing terms)**.** *Syntax for Curry typing adds following type variables and type expressions*

$$V_\tau ::= \tau_1, \tau_2, \ldots$$
$$T ::= V_\tau | T \to T$$

Type statement *of form* $X : \tau$ *where* $X \in \Lambda$ *and* $\tau \in T$ *carries the meaning that the term* X *has type* τ*.*

Declaration *is a type statement where* X *is an atomic variable.*

Basis *is a set of declarations.*

Definition 1.8 (Curry typing)**.** *We say that statement* $X : \tau$ *is derivable from a basis* Γ *in Curry typing, written as* $\Gamma \vdash X : \tau$*,*[9] *if it can be produced by following rules:*

$$(x : \tau) \in \Gamma \implies \Gamma \vdash x : \tau \qquad \textit{(start)}$$
$$\Gamma \vdash X : (\sigma \to \tau), Y : \sigma \implies \Gamma \vdash (XY) : \tau \qquad (\to \textit{elimination})$$
$$\Gamma \cup \{x : \sigma\} \vdash F : \tau \implies \Gamma \vdash (\lambda x.F) : \sigma \to \tau \qquad (\to \textit{introduction})$$

[8] As seen for example in λP system that roughly corresponds with AUTOMATH software [Bru87].

[9] Usually the $\vdash$ symbol is accompanied by some subscript that identifies the typing system used to derive the statement. For brevity we omit the subscripts in all places of this thesis where the system used is clearly identifiable from the context.

Church-style typing is based on a slight change to the syntax of the calculus: instead of λ-terms, we use λ-pseudoterms that carry a type constraint for each abstracted variable – given $\tau \in T$ (same set of types as in Curry case), a λ-abstraction in Church-based typed system that only "accepts" the applied type τ is denoted by

$$\lambda x : \tau.F$$

Theorem 1.7 (Equivalence of Church and Curry typing)**.** *Define a map ϕ to omit the type constraints as*

$$\begin{aligned} \phi(x) &= x \\ \phi(AB) &= \phi(A)\phi(B) \\ \phi(\lambda x : \tau.A) &= \lambda x.A \end{aligned}$$

Then, for each Church-term A,

$$\Gamma \vdash_{\lambda-Church} A : \tau \implies \Gamma \vdash_{\lambda-Curry} \phi(A) : \tau$$

and for each Curry-term B

$$\Gamma \vdash_{\lambda-Curry} B : \tau \implies (\exists C)\phi(C) = B \wedge \Gamma \vdash_{\lambda-Church} \phi(C) : \tau$$

Moreover, the type inhabitation is preserved:

$$(\forall \tau \in T)\big((\exists A) \vdash_{\lambda-Curry} A : \tau \iff (\exists B) \vdash_{\lambda-Church} B : \tau\big)$$

Proof. By induction on structure of A and B. Inhabitation proof is then immediate. □

Forcing the abstractions to carry their type is not particularly convenient for actual programming work – although it has been established as a rule in traditional programming languages (most similar example are the C function type annotations), functional programming languages seldom use any Church-based system. On the other side, type inference in the presence of type annotations is very easy, as the only task left for the inference system is to check if the programmer did not write any conflicting annotations. Programming languages that use pure type systems (discussed later in section 1.3.6) usually greatly exploit this feature.

Following theorem formalizes the most important (and very intuitive) reason of why are the types are useful – if any typed term is β-reduced, its type is guaranteed to stay the same:

Theorem 1.8 (Subject reduction)**.** *If $M \to_\beta N$, then*

$$\Gamma \vdash M : \tau \implies \Gamma \vdash N : \tau$$

Proof. Corollary on the beta-reduction rule. See [BDS13, Proposition 3.2.11]. □

Theorem 1.9 (Strong normalization)**.** *If $\Gamma \vdash X : \tau$ then X is strongly normalizing.*

Proof. There are several methods to prove the strong normalization theorem. Those that extend (weak) normalization can be found in [Klo80]. Method of "computable terms" by Tait [Tai67] was later used to prove the Gödel's *System T* normalization. Usual method to prove the normalization that also extends to System F and several other systems is the one using the saturated sets and the soundness theorem by Girard in [Gir72]. The proof can be found in [BDS13, Definition 4.3.1 to Theorem 4.3.6] □

1.2.2 System F

System F, also known as $\lambda 2$ system, polymorphic λ-calculus or similar, was developed independently by Girard [Gir72] for work with second-order intuitionistic logic, and Reynolds [Rey74] for simplification of functional programming. From the programming side, basic idea for the system is in formalization of the "for each type" statements that are implicitly thought of in simple λ-typing. Consider a term with type

$$\lambda xy.x : \tau \to \sigma \to \tau$$

$\lambda 2$ system notices that, for computation with type systems, a slightly formal problem of whether τ or σ is a free or captured variable might arise, and explicitly captures the fact that the combinator is usable for all types:

$$\lambda xy.x : \forall \tau \sigma.\tau \to \sigma \to \tau$$

Definition 1.9 (System F)**.** *Formally, type language for $\lambda 2$ is modified as*

$$T ::= V_\tau | T \to T | \forall V_\tau.T$$

and the type derivation rules of simply-typed λ-calculus are extended by following rules:

$$\Gamma \vdash A : (\forall \tau.\sigma) \implies \Gamma \vdash A : \sigma[\tau := \rho] \qquad (\forall\text{-}elimination)$$
$$\Gamma \vdash A : \tau \implies \Gamma \vdash A : \forall \sigma.\tau \qquad (\forall\text{-}introduction)$$

For example, I has type $\forall\tau.\tau \to \tau$, Church numerals that work as iterators of any function have type $\forall\tau.(\tau \to \tau) \to (\tau \to \tau)$,[10] and $\omega = (\lambda x.xx)$ can have types $(\forall\tau.(\forall\sigma.\sigma) \to \tau)$, $(\forall\tau.(\forall\sigma.\sigma) \to (\tau \to \tau))$ and $(\forall\tau.\tau) \to (\forall\tau.\tau)$.

System F is strongly normalizing (proof is done in a similar manner as for simply-typed λ-calculus), but simpler variants are used in current programming languages for the purpose of efficient type-checking (decision whether certain term has given type) and typability (whether the term has some type), which are believed to be undecidable for $\lambda 2$. [BDS13, Remark 4.4.1]

1.2.3 Simple Hindley-Milner system

HM system is basically the $\lambda 2$ system with a slight restriction on head-only placement of $\forall$ in type terms. Based on the presence of $\forall$ it then distinguishes *monotypes* and *polytypes*. "Full" HM system as defined by authors also supports several other programming structures and shall be defined later.

Definition 1.10 (Hindley-Milner system)**.** *Type language of HM system is defined by grammar*

$$\begin{aligned} T &::= P \\ M &::= V_\tau | M \to M && (\textit{monotype}) \\ P &::= M | \forall V_\tau.P && (\textit{polytype}) \end{aligned}$$

For $m \leq n$ and terms $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m$, if τ is the result of substitution

$$\tau = \sigma[\alpha_1 := \beta_1] \cdots [\alpha_m := \beta_m]$$

and each β_i is not free in σ, we say that τ is more special *than σ and write*

$$\forall\alpha_1 \ldots \alpha_n.\sigma \leq \forall\beta_1 \ldots \beta_m.\tau$$

Type derivation rules are those of simply-typed lambda calculus extended by following two:

$$\begin{aligned} \Gamma \vdash X : \tau \wedge \sigma \leq \tau &\implies \Gamma \vdash X : \sigma && (\textit{specialization}) \\ \Gamma \vdash X : \tau \wedge \sigma \textit{ is not free in } \Gamma &\implies \Gamma \vdash X : \forall\sigma.\tau && (\textit{generalization}) \end{aligned}$$

Strongest reason to use the HM system is its computational simplicity allowed by the $\forall$ placement restriction: The problems of type-checking and

[10] Second pair of parentheses is not necessary, but makes the fact that the result is again used as a function visible.

typability are solved by Algorithm W (in section 2.2.2) that finds the most general type[11] for any term in time almost linear to the term length (although the problem is actually DEXPTIME-complete, as shown in [Mai90]). Type inhabitation problem is equivalent to the inhabitation problem in simply-typed λ-calculus.

Because most programming languages do not need complex types enabled by $\lambda 2$ (for a good reason), HM has been used many times as a basis for various specific type systems, including those of ML and Haskell.

1.2.4 Intersection λ-calculus

The system $\lambda\cap$ (also known as Torino calculus for strong correlations in the geographical locations of the authors), was introduced mainly by Coppo and Dezani (see, for example, Coppo, Dezani, Barendregt 1983).

Definition 1.11 (Intersection types)**.** *Type terms for $\lambda\cap$ are defined by*

$$T ::= V_\tau | \omega | T \cap T | T \to T$$

Partial ordering $\leq$ on types from T is defined by following axioms:

$$\begin{gathered}
\tau \leq \omega \\
\omega \leq \omega \to \omega \\
(\tau \to \rho) \cap (\tau \to \sigma) \leq (\tau \to (\rho \cap \sigma)) \\
(\tau \cap \sigma) \leq \tau \\
(\tau \cap \sigma) \leq \sigma \\
\tau \leq \sigma \wedge \tau \leq \rho \implies \tau \leq (\sigma \cap \rho) \\
\sigma \leq \sigma' \wedge \tau \leq \tau' \implies \sigma' \to \tau \leq \sigma \to \tau'
\end{gathered}$$

Derivation rules for $\lambda\cap$ are those of simply typed lambda calculus extended by following three:[12]

$$\begin{aligned}
\Gamma \vdash M : \omega & \qquad (\omega\textit{-introduction}) \\
\Gamma \vdash M : \tau \wedge \tau \leq \sigma \implies \Gamma \vdash M : \sigma & \qquad (\textit{generalization}) \\
\Gamma \vdash M : \tau \cap \sigma \iff \Gamma \vdash M : \sigma \wedge M : \tau & \qquad (\cap\textit{-rules})
\end{aligned}$$

[11] Definition: Each type that the expression can have can be specialized to its most general type.

[12] Traditionally the $\cap$-rules are written as two separate implications for introduction and elimination.

Intersection calculus brings many interesting properties. First one is obvious, it allows to state that a combination of properties is expected from a λ-term. Typical example is the typing

$$(\lambda x.xx) : (\sigma \cap (\sigma \to \tau)) \to \tau$$

basically stating that from the programmer's side, the "argument" of the lambda is required to be able to function both as a "primitive type" σ and a map $\sigma \to \tau$, returning τ. Similar property will be used later in this thesis for making some sense in computations of types of ad-hoc overloaded functions.

Using the ω, the $\lambda\cap$ system is able to type all terms, which can bring some system into the situations when programs have to handle otherwise untypable terms. On the other hand, this simplification makes the problem of typability of terms trivially solvable (any term has the ω type). Interesting thing happens in a modified system called $\lambda\cap^-$ that does not include ω type and its rules – not only the strong normalization holds for $\lambda\cap^-$, but the other direction of the implication also holds:

Theorem 1.10 (Strong normalization of intersection)**.** *Term is typable in $\lambda\cap^-$ if and only if it is strongly normalizing.*

Proof. Theorem was proven independently by van Bakel and Krivine, see for example [VB92]. □

1.3 Polymorphism and overloading

Both polymorphism and overloading refer to the concepts where one code has multiple semantic meanings, depending on the context in which it is used. Exact definition varies wildly by programming environment, language and literature. We can use following informal definitions:

Definition 1.12 (Polymorphism)**.** *Term is called polymorphic if more than one type can be derived for it.*

Note that all terms in calculi without atomic type constructors (as defined below) are polymorphic. The distinction made here is useful only for practical languages, where, for example, the logical operators $\wedge$ and $\vee$ are usually not polymorphic, owning only a single typing (bool $\to$ bool $\to$ bool). On the contrary, $(\lambda x.x)$ is polymorphic, with type $\forall\sigma.\sigma \to \sigma$.

Definition 1.13 (Overloading)**.** *Identifier is called overloaded if it possesses multiple definitions, and a different definition may be used at each usage.*

Examples of overloading can not be found in simple λ-calculus since it doesn't allow multiple definitions of the same identifier. Functional languages usually exploit some kind of **let** construction for this purpose, and select the definition to be used based on inferred type.

As there are many algorithms that work correctly on many types of inputs (thus they are polymorphic and the primitives they use to manipulate input data may be overloaded), type systems of modern languages are required to handle such cases accordingly. The idea of polymorphism has already been captured in $\lambda 2$ and HM type systems. For completeness, we will extend the HM system here with the **let** construction and primitive types to allow practical programming and avoid typing of the fixpoint combinator.

1.3.1 Parametric polymorphism in HM system

Lambda calculus is not a very good system to be directly compiled into machine instructions. Consider a case of a function + that is meant to work with integer numbers and has a general type of $\sigma \to \sigma \to \sigma$ (possibly denoted with $\forall\sigma$) – there is clearly some expectation to see the assembly instruction `add` in resulting code, but such compiled function can no longer be used for any other type, breaking the validity of previous type assumption. For this purpose, constant types and type constructors were created.

Similar problems arise with the usage of fixpoint combinator. The "self-generating" code it produces may be actually meant to do very simple things (`while`-like loop or a recursive call) but the reconstruction of the part of the code that actually loops or calls itself requires advanced pattern matching techniques to find the looping code, especially if some parts of the fixpoint combinator definition were already evaluated. Usual approach to this problem is the introduction of **let**-syntax[13] that is meant to handle recursion on its own, making any use of fixpoints unnecessary. More information on handling the recursion in presence of **let** can be found in section 1.3.5.

We will enrich the previous definition of HM system with type constructors and **let**. The new language will be defined by

$$\Lambda ::= V|\Lambda\Lambda|\lambda V.\Lambda|\textbf{let } V = \Lambda \textbf{in } \Lambda$$

and its types by

$$T ::= V_t|C_0|C_i \overbrace{V_t \dots V_t}^{i}|T \to T$$

where C is a set of type constants.

[13] In current programming languages there are several types of `let` operators – here we discuss only the recursion-enabling variant, as `let` from Haskell or `letrec` from Scheme.

Inference rules are extended by

$$\Gamma \vdash X : \tau \wedge \sigma \leq \tau \wedge \Gamma \cup \{x : \sigma\} \vdash Y : \rho \implies$$
$$\implies \Gamma \vdash (\textbf{let } v = X \textbf{ in } Y) : \rho$$

Note that the type generalization is considered only for variables in **let** construction.

By adding the **let**, we have enabled a broad range of polymorphic constructions usable for practical programming. Note that the addition didn't influence the complexity of type inference – the proof from the original HM system still holds.

1.3.2 Overloading

Overloaded identifiers are effectively those that choose their semantic meaning by the context they are used in – such situations usually include calling the same function on various types or using the same constant in different expressions.

The task of choosing the right object to actually use in the expression, usually solved by the compiler or interpreter, is very complex in the naive case when the types of overloaded identifiers may actually be unrestricted. Basic consequence of such language expansion is the loss of the principal type property – the fact that each expression has a single, most general type guaranteed easy manipulation of type expressions in HM and similar systems.

Unrestricted overloading is even proven to be NP-complete. Full proof is done by a reduction to 3-SAT, and can be found in [Pal12].

For this reason, overloading systems are usually constrained on what they can do. A simple example of this is the C++ language, that (omitting some logic on automatic type conversion) states that a pair of functions with the same overloaded name and same types of parameters must not exist, because those would be either indistinguishable (in case the return type would also be the same) or would lead to non-determinism of type assignment if the return types would differ. In functional languages, selection of the overloaded function by return type is quite useful, as demonstrated for example in Haskell – for the simplest instance, observe its variant of "type conversions", usually named like `fromInt` that also elegantly express a constraint on the converted type.

1.3.3 Parametric overloading

Parametric overloading was first fully formalized in the work of Kaes [Kae88]. Basic principle is that all definitions of an overloaded identifier must bear a

type that conforms with the *type scheme* associated with the type. Type scheme itself is a type definition with "holes" that are assumed to have the same type.

Definition 1.14 (Type scheme)**.** *Let \$ be a special symbol not found in type variables. Then ω is called* overloading scheme *if and only if*

$$\omega = \omega_1 \times \omega_2 \times \cdots \times \omega_n \to \omega_{n+1}$$

where each $\omega_i \in T \cup \{\$\}$.

Typical examples of the overloading schemes include

- $\$ \times \$ \to \mathbf{bool}$ for comparison functions
- $\$ \to \mathbf{int}$ for a discrete measure (e.g. list length)
- $\$ \to \$$ for identity functions, successor and predecessor operators
- $\$ \times \$ \to \$$ for binary operators on closed domains $(+)$

Overloading assumptions are then defined to accommodate the identifier that is being overloaded, its type scheme, and the list of types the scheme is applicable to. For example, the overloading assumption for equality test may look like:

$$(=, \$ \times \$ \to \mathbf{bool}, \{\mathbf{int}, \mathbf{real}, \mathbf{list}(\tau_{(=)})\})$$

It specifies that integer, real and list types may be tested, and that the internal type of the list must be again a subject to equality tests.

User-definable parametric overloading

The overloading assumptions may be defined by the programmer. Kaes uses a simple extension of the HM system, where syntax

$$\mathbf{letop}\ x : \tau\ \mathbf{in}\ \Lambda$$

defines a type scheme τ for the identifier x, and

$$y : \tau\ \mathbf{extends}\ x\ \mathbf{in}\ \Lambda$$

assigns an overloaded definition y of *type* τ to x.

Type inference for parametric overloading basically follows the structure of HM system, with several exceptions:

- New syntax correctly sets overloading assumptions for contained code,
- for each identifier, the basis Γ contains the assumptions for all contained types (in the HM system the typing assumptions basically take the form of polytypes, carrying no information about possible application restrictions),
- the usual Martelli-Montanari unification algorithm [MM82] is modified – the step that unifies a variable with occurrence-checked term is extended to also fail in the case when the overloading assumption of the variable does not contain the type that it is being unified with.

Note that computational complexity of the type inference in presence of parametric overloading stays in the same class as for HM system.

1.3.4 Principal type schemes

The possibilities of parametric overloading are limited in many ways, most notably in the presence of single \$ in the type scheme, lack of overloaded constants[14] and absence of *subtyping*, which is defined as follows.

Definition 1.15 (Subtyping)**.** *The fact that the type τ is a subtype of type σ is marked by $\tau \subseteq \sigma$, meaning that for any type term $t(x)$ dependent on its parameter x, $(\forall A) A : t(\tau) \implies A : t(\sigma)$.*

Work of Smith [Smi94] describes a more advanced system that supports both subtyping and overloading in a relatively unrestricted form. Used type system defines a generalized version of type schemes and directly uses the popular notation for parametric type constructors.[15]

Definition 1.16 (Unquantified types)**.** *Given a set of type variables τ_i and type constructors ξ_i, the set of unquantified types is defined as*

$$T = \tau | \tau_1 \to \tau_2 | \xi(\tau_1, \tau_2, \dots)$$

Definition 1.17 (Quantified type scheme)**.** *If τ_i are type variables and $\mathcal{C}_i$ are constraints of the form either $x : \tau$ or $\tau_a \subseteq \tau_b$, then a quantified type scheme σ with the set of polymorphic variables subjected to the constraints is denoted by syntax*

$$\forall \tau_1, \dots, \tau_n \textbf{ with } \mathcal{C}_1, \dots, \mathcal{C}_m \ . \ \sigma$$

[14] Constants can still be overloaded by a non-systematic workaround that produces the constant from unary function with dummy argument.

[15] Generalized and versions of aforementioned dependent types, e.g. the $\textbf{list}(\tau)$.

Given system allows to enrich the typings from the parametric overloading with many new useful constructions.

Simplest example is the function $\lambda fx.f(fx)$: In traditional HM system, its principal type is $\forall\tau.(\tau \to \tau) \to \tau \to \tau$. However, consider the situation when f would be a function that would take a real number argument and return an integer. Although every integer is (or can be easily converted to) a real number and the simple argument conflict would not harm the program logic, the HM typing is unable to express this fact. We will instead embody this information in the assumption $\mathbf{int} \subseteq \mathbf{real}$, and derive a principal type scheme

$$\forall\tau,\sigma \ \mathbf{with} \ \sigma \subseteq \tau \ . \ (\tau \to \sigma) \to \tau \to \sigma$$

which can clearly be instantiated to support f of type $\mathbf{real} \to \mathbf{int}$.

Overloading is expressed using the typing quantifications: Great example is the function $\exp(x, n)$ created for any type that "supports" overloaded multiplication and constant 1. Simple parametric overloading from the previous section is unable to express neither the overloaded nature of the constant 1, nor the need to ensure that the correct overloaded definition is available for more than one required identifier. In the generalized system, we easily construct the principal type scheme that handles this situation:

$$\forall\tau \ \mathbf{with} \ \mathrm{mult} : \tau \to \tau \to \tau, 1 : \tau \ . \ \tau \to \mathbf{int} \to \tau$$

Smith proves that if the typing assumptions are subject to certain restrictions, namely

- if there are no subtypings on types of different "shape", for example $\mathbf{bool} \subseteq \mathbf{bool} \to \mathbf{bool}$,
- if the set of subtypings is not cyclic (i.e. they can be extended to partial ordering) and
- there are no unsatisfiable constraint sets in initial (or user-defined) typing assumptions,

the generalized HM-style type inference algorithm W_{os} for the quantified type schemes as described in [Smi94, Section 3] is again complete, correct and belongs to the same complexity class as the HM system.

The algorithm W_{os} is given for a system where all overloading and subtyping assumptions are predefined, but if some care is taken to ensure that the overloaded identifiers stay global, it is easy to extend to some user-defined form similar to the parametric overloading. Locally overloaded identifiers can be renamed in most (reasonable) cases, but in general they may

introduce difficult situations, force the type system to carry whole overloading information with the typing assumption, solve SAT-like problems and cause undecidable overloading conflicts. According to [WB89], the exact difficulty of the decision in such situation and existence of some easier restriction are both open questions.

Note again the similarity of the requirements with other type systems – partial ordering, global validity of conditions and assurance of satisfiability together guarantee that finding a principal type stays predictable and can not result in complexity explosion.

1.3.5 Haskell types

The type system of Haskell was introduced in [JHA+99] as a practical extension of the principal type scheme system as introduced above, enriched by several new typing rules and syntax – mostly the usage of type classes very similar to their introduction in [WB89] and several concepts not needed for theoretical purposes. There also exists a programmatic, annotated definition of Haskell type inference algorithm (that, again, follows the basic structure of algorithm W), completed by Jones in [Jon00].

Haskell type information is stored in a format similar to quantified types, with the exception that the quantifiers are expressed as general predicates that describe the properties of polymorphic variables.

Resolution of overloading is done using type classes, which are basically lists of predicates (typing assumptions in this case) that each member of the class must satisfy and overload correctly. The syntactical approach, defined by keywords `class` for class definition, `deriving` for expressing class membership and `instance` for assignment of implementations to overloaded identifiers, is sometimes called "dictionary classes" for the fact that the class object only serves as a register of names of overloaded types and their implementations of corresponding class members. Interestingly, as noted in [**?**], type classes do not necessarily have to be hard-coded into a compiler to "work" – Scala language is known to be able to simulate type classes with built-in dictionary objects.

Because the predicates held in quantified type structures usually grow very complex during the type inference, Haskell employs a type simplification method known as context reduction. Its basic task is to convert a set of predicates to equivalent but (by some measure) smaller one. Duplicated predicates, obvious globally valid predicates (like `Eq Ord`), and superclass pairs (`Int a, Num a`) are simply removed. Rest of the predicates is forced to a *variable-first form* – if the predicate states that the type is a member of a class, the first argument of the class definition must be a variable, otherwise

it is converted to one by instancing and adding another predicate (this also creates new possibilities to further reduce the set).

The last important, but probably most striking difference between Haskell and theoretical type systems is the handling of recursion. Descriptions of the original HM system usually circumvent or completely omit[16] the need for typing of recursive variables by adding either a special rule for operator `fix` that is equivalent to the Y combinator and has fixed type $\forall\tau.(\tau \to \tau) \to \tau$, or by adding complex rules for **letrec** or **letrec** ... **and** ... **in** for mutual recursion.[17] Haskell has a special object called *binding group* that roughly corresponds to the notation of multivalue **letrec**, and employs a special careful type inference in that case:

- Explicitly typed bindings are just type-checked,
- implicitly typed bindings are converted to **let**'s where possible,
- mutually recursive function groups are typed in two steps – first "pass" infers the type using the incoming type information separately for each functions, result is then applied again to provide the type checking. There is an important restriction that the recursive functions themselves must not be polymorphic in this environment, otherwise the problem is proven undecidable [Hen93].

1.3.6 Other type systems

There is much functionality that the aforementioned type systems lack, and lots of possible rules and ideas those can not safely express. Usual example of a program that looks intuitively correct but fails to type is that of $(\lambda f.(f(+))(f1)(f1))(\lambda x.x)$ – there is no way to assign both polymorphic types $\mathbf{int} \to \mathbf{int}$ and $(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int})$ to the parameter variable f.

Various extensions to the Church typing[18] have been systematized by the concept of lambda cube [BDS13, Section 5.1], by the kinds of objects that the type system allows to *depend* on others. For example, in simply typed lambda calculus, the abstraction $\lambda x.M$ expects one argument term, which

[16] Several articles leave the recursion handling as an exercise for the reader.

[17] To the best of author's knowledge, the first approach is not very systematic, and all definitions of the second approach that are available in the literature break the let-polymorphism.

[18] Complexity and (non-)decidability of uninformed type checking of complicated type systems usually restricts their practical applicability to Church-style annotated types.

will affect its usage, type and semantics. In other words, the term depends on another term.

In similar generalized sense, the Church variant of $\lambda 2$ system gives terms that depend on types, e.g. in $(\Lambda\tau.\lambda x : \tau.\lambda y : \tau.x + y)$ the type annotation depends on the type of the applied term. Similarly, there is $\lambda\omega$ calculus that adds types depending on types, using *kinds* as "types of types" to ease the construction, and λC system that adds types depending on terms. For details the constructions, this thesis refers to [BDS13, Section 5.4].

Promising research has been done on pure type systems that lift the restrictions of lambda cube by providing an arbitrary choice of type kinds and sorts, effectively creating the possibility to assign a type to anything. Exact definition of PTS is given for example in [RJ07], together with the PTS-based programming language Henk2000 [Roo00], and its type inference and checking algorithms. Roorda also provides an interesting solution to the basic problem of practical programming with pure systems: Explicit Church-style type annotations are unwieldy for programmers, but required for PTS type checking algorithms to stay decidable. The problem is solved by combination of PTS and HM system – only the parts of the program that actually require PTS functionality must be annotated by programmer, the rest is assumed to be "simple" and typed implicitly by a fairly standard HM inference.

2. Compilation of functional languages

Following chapter summarizes the algorithms needed by functional language compilers and interpreters. We will show techniques that cover whole compiler "pipeline" – code representation, type inference, simplification and code generation.

2.1 Parsing and representation

Lambda calculus is, by definition, a context-free grammar. Parsing of reasonable lambda-based functional languages can therefore be performed by a finite automaton with a stack, usually created by one of the standard parsers that fits the compiler programming language – common choices include Bison, AntLR or Parsec (see, respectively, [bis], [ant], [par]). Even the usual syntactic sugar that eases the programming does not require any significant extension of the language.

Target of the parsing process is to get the code to the form that is suitable for running type inference (almost always the next step in the process), that in most cases tightly follows the simple syntax of lambda calculus. For example, intermediate representation of Haskell (after a lot of desugaring) consists only of expressions of following types:

- variables (`x`, `y`, ...)
- data constructors (`Int`)
- literal constants (0, 1, ...)
- lambda abstractions (`\x -> y`, equivalent to $\lambda x.y$)
- lambda applications (`x y`)
- type abstractions (Λ for $\lambda 2$ system, see [BDS13, Definition 4.1.5])
- local bindings (`let ... in ...`, equivalent to the **let** construction demonstrated in chapter 1)
- coercions and casts that were added in later versions to assist subtyping.

Actual method of the term storage depends heavily on the programming language that the compiler itself is written in, but the usual choice is to exploit some kind of runtime polymorphism: In many Haskell compilers, the terms are represented by a polymorphic type in form

```
data Term = Variable str
          | Constructor str
          | IntConstant int
          | Abstraction l r
          | Application l r
          ...
```

C++ implementations tend to use either polymorphic classes and virtualize the common functions, or simulate the polymorphic structures by enumeration.

For purposes of compiler efficiency[1], term storage may be subjected to additional constraints:

- Memory efficiency of storage by eliminating unnecessary duplication of stored terms,
- fast term comparison,
- ease of extraction of various nested information – terms are often queried for their set of free variables or their text representation,
- ease of finding unused identifiers.

Simple techniques that allow to partially satisfy such efficiency requirements on a polymorphic term structure implemented in C++ are shown in the proof-of-concept compiler later in the thesis.

2.1.1 Desugaring

Any language constructions that do not fit in the restricted intermediate representation are viewed as syntactic sugar and must be removed. Most of those constructions are designed to be simple to remove, occupying a similar position as preprocessor macros in C/C++, or `(defmacro)` from Scheme. Functional languages traditionally perform following transformations:

[1] Although compilers are usually not regarded as software where actual performance would be crucial (main area of focus is still the performance of resulting code), more efficiency may indirectly allow for more (highly non-deterministic) speculative optimization. There are also interpreters that must manipulate similar structures as quickly as possible.

- Removal of multiargument lambdas – converting `\x y -> z` to equivalent `\x -> \y -> z`
- Conversion of functions to lambda applications – binding `f x = y` is converted to `f = \x -> y`
- Removal of infix functions, operators and operator constructions.
  - `a ‘op‘ b` gets translated to `(op a) b`, just like `a+b` gets translated to `(((+) a) b)`
  - Partial application of operators, e.g. `(+1)` translates to `\x -> x + 1`
- Pattern matching expressions from Haskell may be translated to `case` expressions
- Bindings in `let` constructions are translated to abstractions.

2.1.2 Common conflicts in syntactical analysis

While lambda calculus is, by definition, free of any parsing conflicts, implementations that allow common infix operators are not. Main problem is the left-associative lambda application rule, which (in a Bison-like parser syntax) may look like

```
application_expr = application_expr other_expr | other_expr;
```

In combination with unary operators or partial operator applications, the absence of any separator between the two halves of the application rule creates many ambiguous situations. For instance, consider the syntax `a -b`, which may be viewed both as subtraction `(-) a b` or application `a (-b)`. Although the parentheses seem to clarify the situation, syntax for `a (-b)` is ambiguous as well, because it may also be viewed as application `a \ x -> ((-) x b)`.

Common priority rules are also a problem: In most languages lambda application has a very high association priority, which leads to counter-intuitive situations where for example `return a+1` means `(((+) (return a)) 1)` and not the (usually intended) `return (((+) a) 1)`.

Solutions for aforementioned ambiguities vary. The tiny language described in this thesis uses special priority rules to sort out lambda application and unary minus priority, with priorities of other possible unary operators (like the `!` for negation) left behind. OCaml authors chose a simpler approach – unary minus is prefixed with `~` to form syntax like `~-1`,

which eliminates the ambiguities altogether. Haskell uses its own variant of parsing and makes the decision about whether operator is unary or binary from existence of whitespace right behind the operator. For example, `(x -y)` is therefore (suprisingly) not equivalent to `(x - y)`.

2.2 Type inference

2.2.1 Unification

Unification of terms is usually the basic element of every type inference algorithm. Usual unification semantics is that of Robinson [Rob65], most implementations are using an algorithm that matches the unification of Martelli and Montanari [MM82]. Although the algorithm is meant to unify logical formulas, lambda calculus can be easily translated to the language of logic – applications, abstractions and all similar constructions are rewritten to predicates in form `apply(x, y)` etc.[2]

Semantic meaning of unification is quite simple: It takes a list of equations of terms that contain variables. If the equations can be satisfied by uniform substitution of some terms to the places of variables, unification returns the most general substitution that can be found (called the most general unifier[3]), otherwise returns failure.

Actual description of unification algorithm can be found in [MM82] or (in many copies) in any textbook of logic programming. We will only add the common modifications for lambda code:

- Checking for equality of large type structures (including long string identifiers) may usually be handled in a single step in reduced (even constant) time, if the term system of the compiler/interpreter supports it. Example is shown later in this thesis.

- Occur check, which is computationally most complex step of unification, may be reduced to a simple operation by caching free variables in terms – this solution would not be appropriate for interpreters or Prolog-like logic programs, but may simplify the situation for compilers.

- Unification steps may be altered for specific purposes – see for example the description of Algorithm Z for type inference of parametric overloading [Kae88, Secton 3.3].

[2] Actually, common intermediate representations of lambda code already have this form.
[3] MGU

2.2.2 Type inference using Algorithm W

Algorithm W is, for its apparent simplicity, described by the set of "natural deduction" rules in almost all presentations in available literature. Because those rely heavily on correct grasp of produced side effects and do not give any serious information about the structures that the algorithm relies upon, this thesis instead rewrites the algorithm to its imperative form, in pseudocode with explicit substitution rules and data structures.

This implementation will also generalize the basic concept of **let** polymorphism, and add a simple[4] type check for the recursive binding-group **let** of following syntax:

$$\textbf{let } v_1 = d_1 \textbf{ and } v_2 = d_2 \textbf{ and } \cdots \textbf{ in } e$$

Primitive types (of constants) are also supported. Their type constructors are expected to be distinguishable from type variables, so that the unification process can treat them just like predicates of formal logic, e.g. the haskell type `List Int` would translate to predicate **List**(**Int**).

Algorithm will use following primitives:

- Substitution type, in our case, is a set of tuples (v, t), meaning that the variable v should get substituted by term t.

- Function $\textbf{subst}(t, s)$ that correctly applies the substitution s to term t by standard definition. If applied on a set, it returns a new set with all elements substituted accordingly.

- Function $\textbf{merge}(s, u)$ that produces a substitution that would be the result of successive application of substitution s and u – result is equivalent to

$$\begin{aligned}\textbf{merge}(s,u) =& \{(v,t) | (v,t) \in u \wedge \neg(\exists t')(v,t') \in s\} \\ & \cup \{(v,r) | (v,o) \in s \wedge r = \textbf{subst}(o,u)\}\end{aligned}$$

- Function $\textbf{unify}(t_1, t_2)$ that returns either the MGU substitution of the terms or a **failure**.

- Function $\textbf{fresh}(\tau)$ that returns a name of an unique new variable that has certainly not been used in any other term so far.

[4]Simple but partially incomplete for careless bindings, see the notice in 1.3.5.

- Function $\mathbf{inst}(\tau)$ that returns an instance of type τ: If τ is a monotype, it is returned right away. For polytype τ the function is defined recursively as

$$\mathbf{inst}(\forall v.x) = \mathbf{inst}(\mathbf{subst}(x, \{(v, \mathbf{fresh}())\}))$$

- Function $\mathbf{close}(\tau)$ that creates a polytype from a monotype with free type variables, by capturing those variables by $\forall$. For example,

$$\mathbf{close}(\alpha \to \mathbf{bool}) = \forall\alpha.\alpha \to \mathbf{bool}$$

Main algorithm function $\mathbf{infer}(t)$ works as follows:

1. Initializes an empty set of substitutions S that will be common for all sub-calls of function $\mathbf{visit}()$, and its purpose is to hold the current type substitution state.
2. Calls $r = \mathbf{visit}(t, \emptyset)$
3. r now contains a set of tuples $(t' : \tau)$ for each subterm of t, where t' is the subterm and τ is its principal type in t. That is the result of the algorithm.[5]

Function $\mathbf{visit}(t, C)$ derives the types for the term t and all its subterms in the presence of typing assumptions from the context C. It is defined for cases depending on the content of term t:

- If $t = \mathbf{c}$ where $\mathbf{c}$ is a literal constant, return $\{(t : \tau)\}$ where τ is the type constructor that describes the type of constant.
- If $t = v$ where v is a variable, then if some $(v : \tau) \in C$, return $\mathbf{subst}(\{(t : \mathbf{inst}(\tau))\}, S)$, otherwise fail because of the unbound variable.
- $t = \lambda x.f$:

  Check if the variable x is not already bound in context C; if it is, replace it by $\mathbf{fresh}()$ and substitute it accordingly in f.

  Call $\nu = \mathbf{fresh}()$, then $r = \mathbf{visit}(f, \{(x : \nu)\})$.

  Search r for the tuple $(f : \rho) \in r$ and return $\mathbf{subst}(r \cup \{(t : \nu \to \rho)\}, S)$

[5]For clarity, we suppose that different terms in the program are also unique for selecting their type from the result set. The semantics is used here only for returning all the sub-results, actual implementation would probably use a side-effect to store the sub-result directly to the term structure.

- $t = xy$:

  Call $(r_x, r_y) = (\mathbf{visit}(x, C), \mathbf{visit}(y, C))$.

  $m = \mathbf{unify}(\nu_x, \nu_y \to \rho)$ where $(x, \nu_x) \in r_x$, $(y, \nu_y) \in r_y$ and $\rho = \mathbf{fresh}()$.

  If m reports a failure, report the unification error and fail; otherwise update $S = \mathbf{merge}(S, m)$ and return $\mathbf{subst}(r_x \cup r_y \cup \{(t : \rho)\}, S)$.

- $t = \mathbf{let}\ v_1 = d_1\ \mathbf{and}\ \cdots\ \mathbf{and}\ v_n = d_n\ \mathbf{in}\ e$:

  For all bindings generate new variable names $\nu_i = \mathbf{fresh}()$.

  Again for each i call $r_i = visit(d_i, C \cup \{(v_1 : \nu_1), \ldots (v_n : \nu_n)\})$.

  For each i call $m_i = unify(\nu_i, \rho_i)$ where $(d_i : \rho_i) \in r_i$. If the unification failed, fail as well; otherwise merge it to $S = \mathbf{merge}(S, m_i)$.

  Finally, call $r = \mathbf{visit}(e, C \cup \{(v_1 : \mathbf{close}(\mathbf{subst}(\nu_i, S))\})$, and return $\mathbf{subst}(r \cup \bigcup_i r_i, S)$.

Note that running all the substitutions everytime the function **visit**() exists is not necessary – same result can be obtained by running the substitutions only when a certain type is actually needed to be fully updated to current substitution, which may save a lot of compiler time.

Also, observe the exact points in the algorithm that actually "cause" the let-polymorphism: Those are exactly the calls of **close**() that mark some type variables as available for replacement, and **inst**() that for each occurrence of the same identifier generates different type "scheme" that matches the type of the context, but is available for unification with completely different types.

2.3 Inlining

Inlining may refer to several concepts – in functional languages it is used as a collective term for various partial operations on the code that are meant to simplify or optimize the program.

Inlining for optimization is actually a difficult problem, because full optimization is hard for any decidable complexity class even in the terms of computational complexity hardness.[6] Most compilers therefore try to target

[6]Consider this "proof sketch": For any problem from a class of decidable problems C and for any problem instance X, we can take a program that solves the problem (no matter what resources it uses), modify it to another program that solves the problem only for instance X by hard-coding the input, and run the full optimizer on it. Optimal program that solves the single problem instance X is of course a program that just prints the solution for X of length n, in time and space $\mathcal{O}(n)$. If the result of full optimizer would be slower, the program would obviously not be fully optimized. By this reduction, full optimization is C-hard.

slightly different metrics.

For our purposes, we will use and separate following techniques:

partial evaluation – when the result of atomic operation of the program can be determined at compile-time, there is no reason not to run the operation right away. Such optimizations include traditional precomputation of constant expressions (e.g. the simple `1+1 -> 2`) and constant propagation (that is able to reduce expressions like `if 0 x y` to `y` or `append x []` to `x`).

partial β-reduction is exactly the β-conversion of available β-redexes (with renaming of fixed variables). This problem is usually rewritten to name inlining (defined below), by a simple conversion to let-form, where for example `(\ x -> y) a` is converted to `let x=a in y`.

inlining of names – by an example, the term `let x = y in x + x` may be inlined to `y + y`. Problem of name inlining is the fact that together with the previous rule, the algorithm basically simulates β-reduction of the calculus, and it may never finish if it tries to optimize a non-normalizing term.[7]

Desirability of name inlining may also be questionable, especially when the substituted term is not simple – consider the code of form

```
let x=y in ... x ... x ... x ... x ...
```

If `y` is big, the situation after inlining would be suboptimal, because the result would contain several copies of the big code piece.

Similar situation may happen if the term y is not big, but just hard to compute. Then, for example, inlining of x in following program would not duplicate much code, but it would duplicate work needed by computation of the compiled program:

```
let x=z in
 let z=y in
  let y=solve_sat_with_500_variables in
   ..x..x...x....
```

[7] β-reduction alone is enough to create unpredictably diverging programs, see section 1.1.3.

There are many measures that help the compilers not to get astray in such situations – only limited number of partial reductions may be allowed per one optimization pass, and some reflection on resulting code quality that will stop unnecessary expansion can be derived from actual code size. This thesis will roughly describe the solution that is used in Haskell compiler. Other concepts include for example heavy non-deterministic optimization as described in [Blu].

dead code elimination – if the variable x is not free in y in term `let x=a in y`, the term is rewritten to y.

parameter dropping and specialization – Whenever parameters of the function are not used in the function body, they can be simply dropped from the function and from all its call sites. This may enable further optimizations (namely dropping the computation of parameter value at the call site) and save some program runtime that was needed to transfer the parameter data.

Similarly, parameter specialization creates a new copy of the function that assumes a fixed value of one parameter – not only it can be dropped, but the fixed value creates new possibilities for constant propagation.

This whole concept may be generalized to lambda dropping (exact reverse of lambda lifting discussed later), which is discussed in great detail in [DS97].

This thesis will further address the two problems of the name-inliner. Following solutions were invented for usage in Haskell inliner and are described in great detail in the report [PJM02].

2.3.1 Termination and loop breaking

Cases when name inlining would never halt are caused by unbounded program recursion that can be detected as cycles in function call graph. It would be easy to never inline *any* function that is known to cause recursion, but results would certainly not be optimal.

Haskell uses a less restrictive strategy – for each inlining pass,[8] it chooses a *loop breaker* as a set of variables such that the call graph contains no cycles if all loop breaker vertices are removed. Those variables are then never inlined. To demonstrate the situation, observe the call graph in a figure.

[8] there is a limited number of inline passes

```
let
  a = ... b,d ...
  b = ... c ...
  c = ... a ...
  d = ... e ...
  e = ... a ...
```

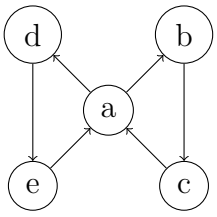


Figure 2.1: Illustration of loop-breaking a recursive definition. Set {a} certainly a loop breaker, but so are the sets {b, d} or {c, d, e}.

Optimal loop-breaker set is chosen heuristically – implementation from [PJM02, Section 4.3] just tries not to select variables that would be beneficial to inline. Actual algorithm involves setting "scores" on how reasonable the inlining of given variable would be (in decreasing order: variables that are simple or atomic score best, variables that occur only once, variables that return a simple constructed type, and all other) and uses any of the best scoring sets.

2.3.2 Code simplification

To see how inlining each variable would affect the quality of resulting code, Haskell inliner uses a simple *occurrence analyzer* that traverses the whole program tree and assigns each binding one of following categories:

Dead if the bound variable does not occur anywhere else,

OnceSafe if the bound variable occurs exactly one time in the code, and its occurrence can not duplicate work, i.e. it does not occur inside a lambda or a constructor,

MultiSafe if the variable occurs more times, but only once in each of several distinct case branches – basically meaning that it still can not be evaluated more than once in the program flow, but the inlining will duplicate code,

OnceUnsafe if the variable occurs once, but in a lambda or constructor, therefore such inline may duplicate runtime work,

MultiUnsafe if the variable occurs more than once, and at least once in a lambda or constructor, so the inlining may duplicate both code and work

OneShot if the variable occurs once in lambda, but the lambda is sure to be called at most once (for example when it is exactly a definition of

OnceSafe or MultiSafe binding). Variable would be otherwise marked OnceUnsafe, but this special case is detected for the reason that it is very common in practice and treating like OnceSafe is beneficial and does not bring any side risk.

LoopBreaker if it belongs to the current loop breaker set.

The inliner then approaches the code schematically as this:

- It certainly inlines all OnceSafe and OneShot bindings, if they are not subjected to external constraints (e.g. are exported from a module).
- It also inlines all trivial bindings – literal constants or variables.
- For other types of variables, it considers separately whether to inline each occurrence of the variable. The decision uses a fairly complex heuristic that depends both on the call site (for detection whether the work would be duplicated) and the type assigned to the binding (mainly for detection whether leaving the code in a shared function would be better). The heuristic itself is out of the scope of this thesis.

The report [PJM02] contains further details about inlining: authors provide a fast algorithm for solving the problem of renaming identifiers, and highlight the argument and measurements supporting why such inlining strategy is simple, fast and very beneficial at the same time.

2.4 Emitting of code

After successful type checking and optimization, compilers are faced with the problem of producing the code for the target machine. We will consider a traditional low-level target, that differs from lambda calculus mainly in three points:

- All code is organized in global-scope functions,
- the functions have a calling convention that accept only fully evaluated arguments of pre-defined types, which directly contrasts with lazy-evaluation of lambda calculus; and
- the code is in a form that is very close to the final assembly – either in some flat language of SSA instructions like LLVM, or just assembly.

This section describes common methods to perform transformations that result in code that satisfies all these properties.

2.4.1 Lifting

Lambda lifting is a fairly standard procedure that takes a "function" – in our case a lambda abstraction or a let-binding – out of its scope, and moves its definition to a global scope. To preserve program semantics, care must be taken not to introduce conflicts of variable names or unbound variables. Algorithm for lifting a was designed originally by Johnsson [DS97, Section 2.2.2]:

- If the converted term is a lambda abstraction that is not a direct subject of the binding[9], produce a new name for anonymous function (for this example "`anon_func`"), and convert the lambda `\x -> y` to let-binding with "named" lambda abstraction:

```
let
  anon_func = \x -> y
in anon_func
```

- Take all free variables in the "definition" of the binding and capture them by adding them to the parameters of function call, and to the abstractions in the function definition. If `a` and `b` were free variables of `y` in the previous example, new function would look like

```
let
  anon_func = \a -> \b -> \x -> y
in anon_func a b x
```

- Functions with fully lifted parameters may be moved to the global scope without any danger, except for possible name collision which is easily solved by globally renaming one of the conflicting functions.

Although full lifting is guaranteed not to change the semantics of programs written in pure lambda language, results on actual computers may vary.

Care must be taken not to enforce evaluation of lifted parameters that would otherwise not get evaluated – for example a branch of conditional code may contain an expression of lifted parameter that either takes a lot of time to evaluate, or does not evaluate at all! Partial solution to this problem is presented in next section.

[9] i.e. it is completely anonymous

Also, transfer of parameters (even when they are evaluated easily) takes time and occupies stack space. Lifted functions with lots of unneeded parameters may not be optimal from this perspective.

Lambda lifting is further examined in [DS97]. Authors introduce an exact reverse process on code blocks, called lambda dropping: While lifting adds (*lifts*) arguments to capture free variables and *floats* the code blocks to global scope, dropping *sinks* the code blocks deeper in the code (closer to their callers) and *drops* the arguments that are captured and applied unnecessarily, if they can be obtained from the scope. Combination of both techniques is demonstrated as a very useful tool for optimization of functional programs, helping to find good candidate functions for possible merging (or generalization) or parameter specialization.

2.4.2 Removing lazy evaluation

As stated above, compiled low-level code can only use strict evaluation. Simple replacement of lazy evaluation with strict evaluation can be a solution in very simple cases, but can cause the program to err or loop infinitely under certain conditions.

To instantiate a situation where lazy evaluation of an expression will terminate, but other evaluation methods are not guaranteed to do so (such expression is equivalent to a lambda term that is normalizing, but not strongly normalizing), consider following function call:

```
f arg1 arg2 (endlessLoop x y) arg4
```

If the compiler decides to evaluate strictly and produce a direct call of `f` with a value produced by the `endlessLoop`, the program will never terminate even if its semantics would be correct (because, for example, `f` would not use the value of its third argument).

Note that deciding termination using the type system would not help in this case – even though in pure lambda calculus the existence of typing in HM system (thus also in System F) would effectively mean normalization of the term, such assumption does not hold for a common language where Church numerals are replaced by integer constants. For example, the code

```
letrec
  a x = if (x < 20000)
        then (a $ (x+1) `mod` 20000 )
else 0
in a 0
```

has a type `Int` in the extended HM system as well as in the extended System F, but is not normalizing, and requires comparatively hard computation to show that it does not terminate.

One possible simple solution to the problem would be to inline the definition of `f` in the code, which could erase (or at least not execute) the occurrence of `endlessLoop`; but because the detection of termination is an undecidable problem, there is no way to reliably tell the inliner which call sites would cause such problems and should therefore be inlined.

Usual functional programming languages workaround the problem by a great margin, by "making lazy values strict": They use *thunks*. A thunk is a description of the value that yet has to be evaluated, stored in some form that the callee function can detect and possibly evaluate. In Haskell implementation, the thunk is a pointer to a memory location where a node of Haskell STG[10] graph that represents the unevaluated call is stored. Since it is a basic property of Haskell evaluation that almost every non-trivial expression is at some point stored in STG in the form of thunk, processing of thunks does not create any serious direct performance impact and can be used for almost every call, except for the case when strictness is directly requested by the programmer.

Main disandantage of using thunks is the fact that they implicitly work as a simple value storage, and they can sometimes hold more values than it would be needed to actually compute the function. Similar behavior can be observed on the simple `foldl` function. Following code listing shows one line for internal representation of each evaluation step:

```
foldl (\x y -> y) 0 [1 2 3 4]
foldl k2 0 [1 2 3 4]        -- name the abstraction for brevity
foldl k2 (k2 0 1) [2 3 4]
foldl k2 (k2 (k2 0 1) 2) [3 4]
foldl k2 (k2 (k2 (k2 0 1) 2) 3) [4]
foldl k2 (k2 (k2 (k2 (k2 0 1) 2) 3) 4) []
k2 (k2 (k2 (k2 0 1) 2) 3) 4
4
```

Observe that even though `k2` lambda is very easy to evaluate, the evaluation is not forced to be called, and at one point a whole new copy of the input list is created on the heap. If similar function would be applied on a dynamically generated (possibly very long) list without any other optimization, result could easily allocate much more memory than expected.

[10] Spineless Tagless Graph-Machine – see for example [JHH+93] for closer description.

Thunks can be generalized for other evaluation methods. Even in a very strict language where the runtime has no possibility to use structures like STG, the delayed call may be easily represented by a thunk of fixed size that consists of function pointer and a list of parameters to pass to the function. Aforementioned `endlessLoop` example can be transformed to a form that can be safely compiled even for a very simple target machine. First, it is possible to produce a modified definition of `f` that accepts a function pointer argument together with parameters of the called function:

```
f a1 a2 resultOfF a4  -->  f a1 a2 ptrToF f1 f2 a4
```

At the call site, the call is then simply transformed to

```
f arg1 arg2 (ptrTo endlessLoop) x y arg4
```

Such transformation is certainly not able to simulate all possibilities of Haskell thunks, especially not for big constructions – the parameters are now stored on the call stack where the space is precious, instead of the STG runtime structure where some allocation does not matter. On the other hand, the full situation is handled at compile-time and the cases where such (possibly endless) stack growth would happen are easily detected by unreasonably big numbers of arguments and excessive stack usage, and possibly either suppressed or reported as error.[11]

Detection of whether it is needed to transform the function to such form of lazy call is still undecidable from compiler side, but the language may add annotations for forced laziness transformation. Similar annotations have been adopted in C++ [Nie08] or Ruby [laz], although their runtime behavior is closer to Haskell allocated thunks.

2.4.3 Code generators

Target code generation is the final step of compilation process. Exact algorithms for code generation of each language may vary widely, but all functional languages usually share a set of builtins and a runtime library which serve as connection points to the target machine, and some code-traversing method that uses the builtins accordingly to the source code and assembles the target code representation. Because of the complexity of the topic, we only give a rough generalization of the process. Details specific

[11] Note that the approach is extremely similar to C++ lambda functions (especially the changes in the callee function), and also applicable to solve partial application of function arguments.

for the proof-of-concept functional language of this thesis are to be found in section 3.2, rest of the information can be obtained from referenced literature.

Builtin objects traditionally serve as a generalized method to translate language primitives (e.g. the most simple functions, operators and control flow constructions) to the target representation. The separation to different named builtin objects has been proven beneficial for compiler structure – using the common "list of builtins", any extension to the language gets simplified to a declaration of new functionality and a method to translate it.

Run-time library is a set of code that is needed to support the functionality of run-time evaluation of language, usually containing library functions (especially the functions that need to be virtualized across platforms, or the definitions of builtins that are too large to inline into generated code), run-time support (in case of Haskell, the functions that govern the STG allocation and garbage collection), or environment initialization (e.g. the allocator initializer in case of C standard library).

There are then several approaches how to perform the actual code translation, depending on how the generation is "driven". We will describe three most common approaches to transform functional code into SSA form.

- Most common method is the "direct translation", that corresponds to code generation of C++, Haskell, Java, or most other languages. The source code is simply traversed as a tree structure in topological order, and each node of the tree is translated to the instructions that depend on the sub-results of its children and position in environment.

  For example, expression `+(f,3)` is translated in this order:

  1. Application of + is found. Builtin for + states that it requires strict values of both arguments.
  2. Call of (parameterless) `f` is found in the first argument. `f` is looked up in the scope, and a call is generated and returned.
  3. Constant 3 is found in the second argument and the target constant code is returned.
  4. Builtin for + generates the code that adds the results of code generation of both arguments and returns it.

- Second common method is the "generator" method, which virtualizes translation by the actual source language – the source code is ran by the interpreter, but the values of the results are not represented as literal values, but as the machine code that generates them.

In a non-existent example language, programmer could then add a definition of how to translate addition to the machine code by following simple statement:

```
a + b = createSSAInstruction "add32" a b
```

Such approach is used in some older compilers of the ML language.

- Third popularized method is the "continuation" method, documented for example in [Tof91] and used in Standard ML language. It reverses some of the process of direct translation method by working only with continuation objects – basically, those are descriptions of what will happen with the program from some point in the code to the end of the program, with defined placeholders (called simply "holes" by the author) for information or values that are not yet known. The program is then inductively generated *backwards*, from `return` or `halt` instruction to the point where only holes are the ones for function or program arguments.

  Authors of the method also claim a great reduction in unnecessary conditional branching of the code, allowed by the fact that the continuations may easily be compared and merged.

Code generators for low-level targets are also expected to break possible infinite recursion that is commonly found in functional languages, usually for the purpose of not running out of limited stack space. The complete and working solution to this problem is the tail-call optimization, described to great detail in any literature concerning compilers. The solution adopted for this thesis is described in section 3.2.6.

3. Low-level functional programming language

We will approach the problem of constructing the main goal of the thesis from two directions:

- Because the greatest problem that prevents low-level usage of current functional languages is their reliance on automated memory management and garbage collection, we will identify the exact language constructions that require this memory management and give proposals on how to replace the functionality in a language without automatic memory management.

- Main result of the thesis is a simple, proof-of-concept compiler of a Haskell-like functional language that produces a code that does not require any runtime support.

In most programs, the managed memory is usually divided into two areas:

- Stack contains a "call trace", with call-local storage of variables and information about where to return after the function exists,

- heap contains everything else.

The separation is most visible in languages that let the programmer manage the heap allocations – the language itself is only allowed to perform automated memory allocation and deallocation on the stack, in a very controlled manner usually defined and restricted by used calling convention.

Resulting possibility for precise control of memory allocation is usually needed to reach certain low-level and high-performance goals. We will therefore seek to obtain exactly this behavior.

In case of functional languages, this separation is hidden. Actual stack contains only local calls of the run-time (of the virtual machine or the automated memory manager), rest of the data is stored on the heap automatically. That includes the actual program stack, which in this case looks more like a linked-list instead of a continuous area of memory. Apart from the automated memory management itself, there are two main reasons to use this structure:

- It can effectively handle lazy evaluation needed for most functional languages, as demonstrated in previous chapter with `thunks`.

- It handles the situation when an object of non-predictable size must be stored on a stack.

  For example, consider a function that would want to return a first-class list object using a C-like calling convention on a traditional stack. Even if we ignore the architectural limit on stack size (usually restrictively small), none of the caller and callee functions know in advance what will the size of the object be – the caller can therefore not "reserve" any deterministic amount of space for the object, and the callee can not be counted on to "somehow move and rearrange" the object to the caller's stack space along with size information – such behavior would be both very complex and slow.

  Simple observation reveals that objects with unpredictable run-time size that require this behavior are exactly those that possess a recursive definition, e.g. the lists, trees or strings.

Next section shows the possibilities of replacement this functionality on a language that is constructed by removing the automated memory management from Haskell.

3.1 Reduced Haskell

For illustration purposes, we will simply assume a language similar to Haskell, but *without any recursive types*, i.e. without the possibility for the type to (directly or indirectly) reference itself. As stated before, that effectively guarantees that the stack memory of the language is predictable and can be subjected to the conditions similar to those of low-level languages.

For example, following construction is forbidden in the reduced language:

```
data List a = ListItem a (List a) | Nil
```

In such language, traditional functional constructions are impossible, but many effective workarounds exist and are already widely used in the programming world. We will show that the language can safely accommodate *pointers* and encapsulated *object constructions* as known from C++ to easily simulate the lost functionality. Discussion about replacement of lazy evaluation and partial application follows after that.

3.1.1 Types and pointers

Implementation of actual pointers to the reduced language is easy. One can extend the IO monad with two simple methods (preferably with some more compact syntax):

```
readPtr :: Ptr a -> IO a
writePtr :: a -> Ptr a -> IO ()
```

In a similar way, standard calls for `malloc` and `free` from the C runtime may be wrapped.

Recursion in the types can then be replaced by simple usage of pointers. The singly linked list from the above example would then be defined as

```
data List a = ListItem a (Ptr $ List a)
```

The implicit enumeration (by operator `|`) is not necessary in this case, as Nil list can be easily identified with a null pointer), but may be useful in other cases and has to be replaced too.

Enumerated types

Implicit enumeration itself can be replaced as well – common approach to the problem is to number the possibilities and add a simple integer tag to the type that will be matched on operations that require access to type structures (e.g. the language pattern matching); and union all "contained" data.

The construction is simple, gives a natural-looking way to common definitions (for example the definition of `data Bool = False | True` then exactly replicates how the actual binary values for False and True look like in other languages), but may present a problem in situations where the integer tag is unacceptable. Such cases must either be "compiled out"[1] or the type must be defined by hand, without `|`.

3.1.2 Partial application and lazy evaluation

We will demonstrate the concept on following code:

```
(if (cond) then functionA else functionB) param1 param2
```

In the reduced Haskell, there is obviously no possibility to actually return the function identifier or function definition lazily – both would require (impossible) run-time name resolution. We will therefore need to come up with another solution.

First, observe that partial evaluation could eventually come up with following code:

```
if (cond) then (functionA param1 param2) else (functionB param1 param2)
```

[1] This phrase is used for the code that is partially evaluated or inlined to the state where it effectively disappears.

Still, such case requires a generalization of the concept of `if` and addition of complex derivation rules to the partial evaluator.

Better approach is to handle the situation exactly like C-like languages: We can follow the example construction from section 2.4.2 – generate function pointers, process them with `if` (choosing one of them), and call the function that is referenced by the resulting pointer.

That concept needs some construction in the language to be applicable to general partial evaluation. We will call the result *static thunks*, which can be thought of as structures with stored function pointers with some values of the function parameters that are pre-set. Note that such object is of fixed-size[2] and non-recursive, therefore it can be safely stored on a stack.

The compiler must be adapted to support automatic construction of such thunks on cases of lambda application. Thunk usage concerns following cases:

- Identifier with a name of function is simply translated to a value with the function pointer (which is now considered a static thunk). From the perspective of the typing system, this has the original type of the function.

- Thunk applied to a value is simply expanded to hold the value. Typing system guarantees that the value type is correct for later application.

- Thunk applied to a thunk must be flattened to a single thunk – compiler may produce an anonymous function, that

  - accepts all parameters of both thunks and the unfilled parameters of the "outer" thunk,
  - on call, reconstructs the value of the "inner" thunk from the parameters,
  - calls the "outer" thunk function on the reconstructed parameters;
  - resulting thunk holds a pointer to the newly generated "proxy" function.

  Because of the fact that the code for each function must be generated in many versions for all possible different thunk applications that are used in the program, this approach may generate a great amount of almost-duplicate or unnecessary code. Still, most of it (especially the small proxy functions) can be either inlined or compiled out.

[2] fixed for each occurence of partial application

- Thunk that already has all parameters (that can be detected from its type) can be transformed to actual call when its strict evaluation is needed.

Effectivity of the whole process will be directly dependent on the performance of the optimizer that is expected both to partially evaluate or compile out most of the pointer-indirections in the code and, possibly, to generate just the needed thunk-accepting function versions.[3]

Because it is automatically indistinguishable whether given code can be evaluated strictly and still terminate, the annotated-laziness may be a good option for the possible implementation: The programmer will be able to insert laziness wherever needed, and will be also able to control the amount of static thunks generated.

3.1.3 Linking to other languages

With proposed execution model, linking of reduced Haskell functions with other languages is extremely easy. The set of requirements for linking is similar to those of C++ or Haskell FFI[4]:

- Call convention must be the same on both linking sides
- Function types must not be polymorphic and should be unified on both sides of linking
- There must be some functionality that marks the function as exported or external, together with exact type of the function

Simple (although not very complete) example of the linking with foreign language is given in the proof-of-concept compiler:

- The `main` function is exported as symbol with the same name and gets linked from the C runtime,
- functions `printf` and `scanf` are linked from the C runtime (a simple wrapper that fills the arguments is used),
- generally all functions defined in the program can be directly called from external code using the `fastcc` calling convention, with their types simply rewritten to traditional declarations. For example, `incr :: Uint -> Uint` would have C declaration `uint64_t write(uint64_t)`.

For further details, we refer to section 3.2.4.

[3]This may beneficially correspond with the tendency of most compilers to optimize code size.

[4]Foreign Function Interface

3.1.4 Destructors

Implementation of automatic destruction of allocated resources is a crucial point that enables comfortable manipulation of complex data structures within the language.

Reduced Haskell inherited the traditional constructor functions from Haskell – those may allocate resources (open files, grab locks, occupy memory), but without the garbage collector there is no way how to free those resources automatically anymore.

C++ approached this problem using destructors: Before some space for an object is deallocated (i.e. either removed from a stack, unrolled or `delete`'d), the compiler automatically calls a special function that is meant to free all held resources.

Haskell approaches the problem from a higher level – all resources that eventually will have to be released (i.e. all possible resources that are in danger of being "held" unnecessarily) are recognized by the garbage collector that is able to automatically and correctly release them. That is convenient, but limits the diversity of resources that the language can possibly manage automatically, as the destruction routines are usually hard-coded into the compiler. Moreover, indirect result of this automatic deallocation is the fact that there is now no universally-accepted syntax or semantics for destructors in Haskell-like languages.

There are several possibilities how to replace this functionality in the reduced Haskell:

- To call the destructors by hand. That approach is prone to programmer error (who would simply forget to call the destructors at all exit points) and unusable in the presence of exceptions, where stack unrolling is needed.

- To add the destructor calls by language convention. This is exactly the example of C++, which implicitly adds a destructor call to each exit point of the function, as well as to the unroll plans for exception handling. Basic problem of this approach in functional programming is that the "exit point" of the function is not explicitly defined.

  Obviously, every state change is contained in a monad. It would be therefore logical to add the destructors to the ends of lexical monads, and provide an overloaded `destroy` operation for all types that require destruction.

  ```
  func = do
    a <- allocResource   --detect binding as allocation
  ```

```
...
destroy a    --implicitly inserted by the compiler
```

Basic problem is the nature of the monads: They can be also defined without the `do` syntax (using e.g. only `>>=` operator) so it would be necessary to guess where the defined "submonad" actually ends. One of the basic properties of monads is the fact that they are monoids, in our case that `do { do a; b} ; c` must be equivalent to `do a; b; c`, which effectively says that we can not find any other end of the monad than the end of the program.

Lexical validity rule, destruction of the variable when it is sure to no longer be in scope does not apply to functional languages for a similar reason.

- Haskell library provides an interesting option in the implementation of `bracket` – the syntax basically serves as a framework for construction, usage and destruction procedures on some allocated resource. Consider the example from the `bracket` documentation [bra15]:

```
bracket
  (openFile "filename" ReadMode)    --constructor
  (hClose)                          --destructor
  (\fileHandle -> do { ... })       --work with the fileHandle
```

The syntax explicitly avoids the aforementioned monad-ending problem. Moreover, `bracket` also catches the possible exceptions and simulates unrolling by correctly calling `hClose` in case of problems and can be easily embedded in specialized construction or destruction routines.

Note that similar problems as with destructors arise also with possible copy constructors, which are usually avoided in any functional programming as being completely irrelevant. Possible future implementation of the reduced Haskell will have to either circumvent the situation (possibly using a level of indirection by copying "references" as in Java or Python), or come up with a novel solution or syntax for such problem.

3.2 Implementation

This section describes the implementation of the main result: The proof-of-concept compiler of a low-level functional language is coded in C++, works on

any modern UNIX platform that supports the dependencies, and is attached to this thesis.

C++ was chosen partly for author's familiarity with the language, partly for the fact it supports standard language parsers, and partly for easy integration with LLVM framework that is used to process the resulting low-level code into target-specific assembly.

A practical introduction to the usage of the compiler is given in the Appendix.

The language itself is a small, Haskell-like language with several simplifications and some resulting limitations. List of the known surprising limitations is given in section 3.2.8.

3.2.1 Syntax

Syntax of the language is defined in file `parser.y`, but best explained on the source of above `loop` example:

```
1.  main = loop 0 10 [
2.    loop :: Uint -> Uint -> Machine;
3.    loop i max = {
4.      write i;
5.      if (i<max) (loop (i+1) max)
6.      (return 0);
7     }
8. ]
```

Line 1 contains the definition of the main function.

Let-syntax is provided in a similar way how the substitutions are usually marked in the lambda calculus – `a [b]` means exactly `let b in a` from Haskell. Note that definitions must be separated by a semicolon – we do not allow the indent-driven separation syntax.

Main function contains a single call of the loop function with parameters.

Typing of the internal function on line 2 is not necessary, but was added for clarity and demonstration.

Curly braces { } are used for the same purpose as Haskell `do` notation, they basically transform the inside functions to a monad syntax using the `>>` and `>>=` operators (which are supported too).

Conditional execution with `if` of a simplified syntax either loops with another call of the `loop` function, or uses the `return` to assign a zero value to the monad.[5]

[5] `return` here is exactly the monadic return, and does not anyhow terminate the function execution. In this case, it just accidentally terminates right after.

3.2.2 Syntactical analysis

Syntactical analysis is carried simply by the usage of Bison and Flex – the token scanner is in usual file `scanner.l` and the parser in `parser.y`.

The program uses a fairly standard method of parsing to an AST structure defined in `ast.h`, that is in second phase (for purposes of simple checking and desugaring) transformed to actual intermediate representation. The second transformation is defined in the file `ir.cpp`.

3.2.3 Term storage method

The compiler demonstrates several possible improvements on term storage that were mentioned in section 2.1. Term structure is defined in file `term.h` – from the programming perspective, it contains an enumerated value for the type (integer constant, identifier, lambda application, ...) and pointers to the sub-terms, or possible values of the constants (integers, strings).

Basic improvement over the simple storage method is the fact that each term is stored only in one unique place – two terms with the same type, subterms and data are *guaranteed* to share their memory location. This is achieved by storing the terms in a splay tree that is ordered "lexicographically" on terms – by the term type, then by the term value (if it exists), and subsequently by the lexicographical list of the subterms. Already stored terms are then never modified on any variables that could break the ordering, and the program only uses pointers to refer to them.[6]

Technically, the splay tree storage is achieved simply by adding three term pointers `LeftChild`, `RightChild`, `Parent` to each term and maintaining correct splay tree structure.

Such storage brings following new possibilities:

- Terms of *arbitrary size* that are saved in the splay tree can be quickly compared for equality simply by comparing their pointers.
- Saved terms may be compared lexicographically in amortized time[7] $\mathcal{O}(\log n)$, simply by splaying the first term to the root, splaying the second just after, and detecting the direction of the last tree rotation[8]. Note that this might seem slow, but the same algorithm works equally on terms of arbitrary size, where simple lexicographic comparison would need to traverse the whole term structure. The

[6] Adhering to this rule also prevents accidental construction of self-referencing infinite term.

[7] in this analysis, n will designate the number of already stored terms

[8] If the second term "came from the right", it was obviously greater, and vice-versa.

function `term::ttree_cmp` that performs this comparison can be found in source file `term.cpp`.

- If at least one of the terms is not stored in the splay tree, but all their subterms are already stored there, they can be compared in amortized time $\mathcal{O}(s + k \log n)$, where s is the time needed to compare the data stored in the terms (e.g. text strings with variable names), and k is the number of subterms of the term with less subterms. Algorithm simply compares the term data, and then uses the previous in-tree comparison to all subterms, returning the first non-matching result, or signaling equality if no mismatch was found. The function that implements this functionality is named `term::cmp`.

- Finally, saving a term to the tree is performed by following operation:

  1. A random term from the tree (preferably the last one saved[9]) is splayed, to "find" the tree root for binary search.
  2. Term pointer `cur` is initialized to point to the newly found of the tree, and two empty sets of term pointers `left` and `right` are created.
  3. Algorithm produces a decision on how to continue with the tree search:
     (a) If `cur` is in `left`, move to the left children.
     (b) If it is in `right`, move to the right.
     (c) Compare the to-be-saved term with `cur` using `term::cmp`. If they match, return `cur`.
     (d) If the `cur` is lexicographically less then the new term, add cur to the `right` and continue right. If the term is lexicographically greater, do the same operation with exact opposite sides.
  4. Repeat previous steps until a free leaf is found.
  5. Save the new term to the leaf, splay it, and return a pointer to it.

  It has to be proven that the algorithm indeed terminates – there is certainly a risk that the splaying involved in term comparison of the subterms might splay the `cur` node and the search would therefore cycle forever. If none of the terms compared in the step contain `cur`, there is zero risk of any damage to the search operation – `cur` is not

[9] Last used term is supposedly "hot" and near the actual splay tree root.

moved up, it may only be "splayed through" by its child that is splayed, which actually lowers its children count (and therefore speeds up the algorithm). If `cur` happens to be a subterm of any of the compared terms, it may be splayed, but the sets `left` and `right` remember the decision for each of the compared terms, and cycling forever is therefore impossible.

Actually (and in the average case), the situation is lot better – probability that the exact term is a subterm of one of the involved terms is extremely low and the situation of the cycling terms usually happens only in the "root area" of the splay tree, getting quickly sorted out using the cached comparisons. Average complexity is then kept near $\mathcal{O}(k * \log n)$.

Experimental measurement obtained from collecting the search cycle repetition statistics from example program compilations, as shown in the plot, statistically support the complexity claim.

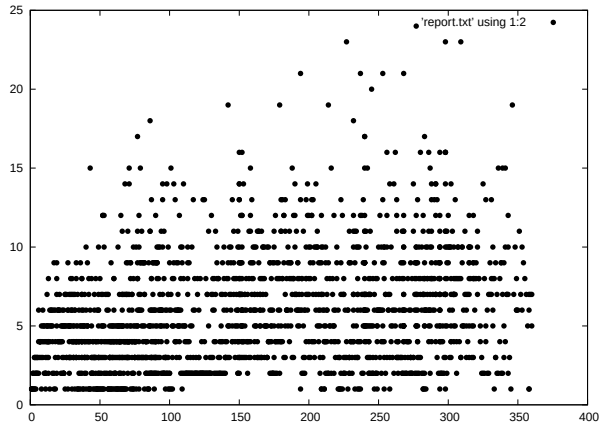


Figure 3.1: Scatter plot of the count of iterations in `term::save`. Horizontal axis is the size of the whole tree, vertical is the count of iterations.

The fact that the terms are stored in a constant structure may be further exploited to precompute or cache interesting or useful information about the terms. Program uses this to cache a formatted, string form of the term (see function `term::format()`), but for specific purposes, more information can be cached with almost no cost for the compiler performance. Specific example of caching the free variables could be implemented in a similar way.

There are also other simple tools associated with the term structure – the one that has been proven most useful is the transformation pass framework defined in `pass.h`: The pass() function allows to simply traverse and modify any information in the term, and is defined as a DFS-ordered tree visitor that saves some temporary environment information for each visited subterm and provides an easy interface where any transformation pass can be constructed using two simple callback functions (one for entering and one leaving the term in the subtree) that only modify the local environment. Simplest example usage can be seen in the `inline.cpp`, where it is used to quickly remove all type information from the program before inlining.

3.2.4 Code generator

TinyL code generator follows the "direct translation" model presented in section 2.4.3 – it receives the properly lifted and inlined code and is expected to produce a LLVM module that contains the translated functions.

First, the (small) runtime is inserted in the module, so that the builtins have all the requirements to work correctly, which currently includes declarations of external functions and preparation of some constants. The task is done by function `builtin_prepare_globals()` in `builtin.cpp`.

In a similar manner, function prototypes of all defined functions are inserted into the module, for the purposes of possible future reference (and recursion). This is done using function `codegen_func_proto()` in `codegen.cpp`.

After that, main code generation can start. LLVM framework stores generated code into blocks represented by class `llvm::BasicBlock`, and uses a simple builder abstraction that correctly inserts instructions into given block. Code generation is simplified thanks to the fact that the result (actually a reference to the code that generates the result) of each operation may be easily stored in form of the `llvm::Value*` pointer, making the binding of successive instruction calls very easy.

The code generator itself, found in function `codegen_code()` is not very complicated: The pass framework (described above) is used to hold the current code block, builder and scope, to allow generation of the code for various code branches. Code generation itself is then rather straightforward:

- If the generator finds a constant, it generates a LLVM constant,
- if it finds a variable, it takes its value from the scope,
- if it finds a function call, it recurses to build code (and obtain LLVM Values) for the parameters and creates a call instruction, and

- if it finds a builtin, it lets it direct the recursion of pass function, using the callbacks for `enter` and `leave` specific to the builtin. The exact function and structure of a builtin is discussed in next section.

Resulting LLVM module is then dumped in the bytecode file, and compilation is finished.

3.2.5 Builtins

All non-core language features are meant to be easily separable and extensible, and are therefore contained and implemented in builtin objects, in file `builtin.cpp`.

A builtin is represented by the struct `builtin_def` that is usually referenced by terms of type `BUILTIN_LAMBDA`. Such terms have roughly the same meaning as an identifier with reference to some function, but instead the evaluation and code generation is handled by a code specialized for the builtin.

The builtin definition structure consists of following information:

- Builtin name string (e.g. "+" or "if"),
- builtin type and arity (e.g. the "+" has type `Uint -> Uint -> Uint` and arity 2),
- pointer to the function that does partial evaluation,
- pointers to the functions that handle code generation of the builtin, namely the `cg_enter` and `cg_leave`.

Builtin name and type are used in the beginning of the compilation process, when correctly named and typed definitions of the builtins must be injected into the loaded code. For example, we present a code for function that generates the type `Uint -> Machine` for the `write` builtin:

```
static term_ptr type_write (ttree&TT)
{
  term i, c;
  i.type = term::IDENTIFIER;
  c.type = term::TYPE_LAMBDA;
  i.str = "Uint";
  c.a = i.save (TT);
  i.str = "Machine";
  c.d = i.save (TT);
  return c.save (TT);
}}
```

Note also the example usage of term object and term pointer creation – the `ttree` is simply a container that holds the term objects so that they

do not get accidentally reallocated, and there is no trouble with eventual deallocation (or garbage collection) of the term tree. Struct members `.a` and `.d` are the left and right "operand", named like this for same reasons as the `car` and `cdr` from Scheme.

Partial evaluation function of the builtin is called in the inlining pass, where it tries to evaluate the (supplied) term arguments or just transform the code to something simpler. For demonstration, we present the definition of partial evaluation of operator "$", which serves just as a shortcut for parentheses, and for "+":

```
static bool
dollar (term_ptr&t, ttree&TT)
{
  term n = *t;
  n.a = t->a->d;
  //n.d = t->d is already set
  t = n.save (TT);
  return true;
}
```

```
static bool
inline_plus (term_ptr&t, ttree&TT)
{
  if (t->a->d->type != term::UINT
      || t->d->type != term::UINT)
    return false;
  term n; n.type = term::UINT;
  n.uint=t->a->d->uint + t->d->uint;
  t=n.save(TT);
  return true;
}
```

The code-generation directing functions of the builtin, `cg_enter` and `cg_leave` are virtualized to let the builtin decide how should the result be arranged in the resulting code structure.

For example, the definition of `if` certainly needs to recurse to the first parameter, but the second and third *must not be unconditionally evaluated* in the same code block. Therefore, the `enter` function of `if` runs the code generation with a new code block and builder for both branches, and the `leave` function correctly creates the conditional jump to the new blocks, return jumps from both branches to the completely new "merge" block, and a phi-node to retrieve the result of the whole `if` instruction.

Here, simpler functions are demonstrated for completeness: A generic `enter` function for strict evaluation of parameters, and a `leave` function for "+" operator.

```
static void
cg_stricteval (term_ptr self, pass_env<codegen_data>&env, Module*mod)
{
  term_ptr t = &env.t;
  codegen_data nd (env.data() );  //clone the environment
  while (t->type == term::APPLICATION) {   //recurse for all applications of the call
    env.data().argspace.push_front (NULL);   //create a space for the result
    nd.save = &env.data().argspace.front();  //mark a point for saving the result
    env.visit (t->d, nd);   //tell the environment to visit the argument
    t = t->a;    //go to next argument
  }
}

static Value* cg_plus (term_ptr self, pass_env<codegen_data>&env, Module*mod)
{
```

```
  std::vector<Value*> args
    (env.data().argspace.begin(),
     env.data().argspace.end() ); //copy the arguments from the environment
  return env.data().builder->CreateAdd (args[0], args[1]); //emit LLVM instruction
}
```

Function `CreateAdd` is the actual LLVM call to create an adding instruction. One can also observe the function of `pass_env` structure, and the (delayed) visit of the pass framework.

All builtins are finally registered in an array of builtin structures, and are ready for usage in the environment. Complete and final builtin registration to the array is shown in the last example:

```
static builtin_def builtin_list[] = {
  {"+", 2, type_2uint, plus, cg_stricteval, cg_plus},
  {"-", 2, type_2uint, minus, cg_stricteval, cg_minus},

//....

  {">>", 2, type_proceed, no_inline, proceed_enter, proceed_leave},
  {">>=", 2, type_bind, no_inline, bind_enter, bind_leave},
  {"", 0, NULL, NULL, NULL, NULL} //last entry
};
```

3.2.6 Handling recursion

As stated above, the program must handle the cases when infinite recursion would allocate stack space with no bounds. Solution using a tail recursion in TinyL compiler is extremely simple and effective – as there are not any complicated allocated resources, any function calls can simply be rewritten to tail calls by the code generator. LLVM can give us even more flexibility in the choice of marking a function call either `tail` to enable tail recursion optimization, or `musttail` to *require* the tail recursion. The requirement may be useful in actual programming environment where programmer is sure to use some functions as loops. In that case the function call would get marked as `musttail` in the language, which would translate to the same LLVM requirement and produce an error if the call could not be tail-optimized, to prevent possible stack depletion in runtime.

We give an example of the LLVM bytecode[10] and resulting assembly code of the program `rw.tl`, that simply reads the numbers on the input and writes some simple modification of them, until a zero is entered.

The source of the program is following:

```
writer x = {
```

[10]Representation in the human-readable form of the LLVM language can be obtained from any `.bc` file by using the `llvm-dis` tool, usually shipped with the LLVM toolchain.

```
  a <- read;
  write (a*x);
  if (a!=0) {writer (x+1)} {return 0};
};

main = writer 0
```

In the figure, notice the result of lifting the anonymous function (that was created by monad-syntax rewriting in the `writer` definition) and the `tail` marks on the calls in the resulting LLVM code.

Translation to assembly results in final code shown in the second figure (again stripped of information that are not interesting for this demonstration). Notice that all "looping" calls were rewritten to actual jump instructions, as they all appeared only in tail calling positions.

3.2.7 Compilation overview

The overall compilation process may be observed in `main.cpp`. After the source is loaded and parsed to intermediate representation,

- the root term is injected with builtin definitions,
- it is type-checked by algorithm W (that can be found in file `infer.cpp`),
- type information is discarded from terms and names are inlined (see `inline.cpp`). Currently, inlining is not very invasive – only the terms that are sure not to do any harm are inlined, and builtins are allowed to process the code (if they are able to);
- all lambda abstractions are converted to named functions, (`lift.cpp`)
- all code blocks are lifted and dead code is eliminated, and finally
- the code generator (in `codegen.cpp`) creates the LLVM bytecode for each function.

Output of the last step is saved to the file with extension `.bc` and is expected to be processed by LLVM assembler and linker.

3.2.8 Possible extensions

As stated above, there are many limitations and simplifying design decisions the compiler was subjected to, in order to stay small and fit the purposes of this thesis.

Several limitations of the compiler may be surprising, especially for the users of Haskell language:

```
define internal fastcc i64 @writer(i64 %x) {
entry:
  %0 = alloca i64
  %1 = call i32 (i8*, ...)* @__isoc99_scanf(.....)
  %2 = load i64* %0
  %3 = tail call i64 @_lift__anon_func_3636(i64 %x, i64 %2)
  ret i64 %3
}

define i64 @main() {
entry:
  %0 = tail call i64 @writer(i64 0)
  ret i64 %0
}

define internal fastcc i64 @_lift__anon_func_3636(i64 %_liftvar_x36, i64 %a) {
entry:
  %0 = mul i64 %a, %_liftvar_x36
  %1 = call i32 (i8*, ...)* @printf(.....)
  %2 = zext i32 %1 to i64
  br label %proceed

proceed:
  %3 = icmp ne i64 %a, 0
  %4 = zext i1 %3 to i64
  %ifcheck = icmp ne i64 %4, 0
  br i1 %ifcheck, label %then, label %else

then:
  %5 = add i64 %_liftvar_x36, 1
  %6 = tail call i64 @writer(i64 %5)
  br label %ifcont

else:
  br label %ifcont

ifcont:
  %ifval = phi i64 [ %6, %then ], [ 0, %else ]
  ret i64 %ifval
}
```

Figure 3.3: Example `rw.tl` translated to LLVM low-level code

```
  .type  writer,@function
writer:
  pushq  %rbx
  subq  $16, %rsp
  movq  %rdi, %rbx
  leaq  8(%rsp), %rsi
  movl  $.Linfmt, %edi
  xorl  %eax, %eax
  callq  __isoc99_scanf
  movq  8(%rsp), %rsi
  movq  %rbx, %rdi
  addq  $16, %rsp
  popq  %rbx
  jmp  _lift__anon_func_3636

  .globl  main
main:
  xorl  %edi, %edi
  jmp  writer
```

```
_lift__anon_func_3636:
  pushq  %r14
  pushq  %rbx
  pushq  %rax
  movq  %rsi, %r14
  movq  %rdi, %rbx
  imulq  %rbx, %rsi
  movl  $.Loutfmt, %edi
  xorl  %eax, %eax
  callq  printf
  testq  %r14, %r14
  je  .LBB2_1
# BB#2:  # %then
  incq  %rbx
  movq  %rbx, %rdi
  addq  $8, %rsp
  popq  %rbx
  popq  %r14
  jmp  writer
.LBB2_1:  # %ifcont
  xorl  %eax, %eax
  addq  $8, %rsp
  popq  %rbx
  popq  %r14
  retq
```

Figure 3.4: Resulting assembly code of program `rw.tl`

- The foremost limitation is the used type system, which is only a simple HM system that is extended with the multi-letrec construction as described in Chapter 2. This allows the compiler to run a complete type check on the whole code, but limits some of its basic features. For example, because the system does not support overloading, the main monad Machine (replacement of Haskell IO monad) allows only the constructions that would be otherwise marked as Machine Uint. Uint here represents a 64-bit unsigned integer. The compiler effectively does not support any other type yet.

- Function `main` is forced to have type `Machine`, and is automatically compiled and exported in the module as symbol `main` – from that point, the compilation process may continue with linking exactly as in the case of C-like languages.

- Evaluation of all function calls is strict.

- Because of the problems with polymorphic recursive bindings (see the notice in section 2.2.2), some of the basic builtins are not polymorphic in the code, unless explicitly redefined in a let-construction. Direct outcome of this fact is that it is hard to use more than one `$` operator in whole code – derived types usually do not unify in the non-polymorphic environment of the root `letrec`.

Possible future work on the compiler should certainly improve following functionality:

- Include more builtins. Writing builtins is easy and directly extends the possibilities of the compiler.

- Provide compound type constructors that would allow to construct structures by hand.

- Provide an overloading- and recursion-capable type system, mainly to support the polymorphic Machine monad.

- Implement better and more aggressive inlining, in combination with some of the solutions for lazy evaluation.

- Allow partial application by using the lazy evaluation solution from section 3.1.2.

Conclusion

This thesis has explored current possibilities of implementing compilers for low-level functional languages.

The ancillary result of the thesis is the summarization of knowledge that is required to implement such compilers. Theoretical requirements for work with functional languages, review of some properties of lambda-based languages and examples of several typing systems were collected in chapter 1. Complete chain of algorithms needed for whole compiling process was discussed and presented in chapter 2 in a straightforward form, with the intention that the reader would easily understand the compiling process of most of current functional languages.

Main result of this thesis is the construction of the proof-of-concept compiler for a simple functional language, called TinyL. The implementation contains

- a complete parser for a Haskell-like functional language,
- a method of term storage with several interesting properties (see section 3.2.3),
- type inference and checking system (the standard algorithm described in section 2.2.2),
- several (very simple) optimization methods,
- lambda lifting implementation,
- a code emitter that transforms the code into LLVM low-level instructions (see section 3.2.4) that are then used to produce a working binary.

The main difference from other current functional languages is that the compiled code does not require any runtime support; specifically, the produced code works without automated memory manager or garbage collector. Resulting execution and memory model is roughly equivalent to that of C language. The code can be easily linked with modules written in other languages – although current implementation does not support that directly, the possibility is demonstrated by linking with standard C library to use its program initialization code and several library functions.

The implementation of TinyL compiler was discussed in detail in chapter 3, providing an insight into programmatical aspects of compiler construction

and describing properties of internal data structures of the compiler. A practical introduction to work with the compiler is given in the Appendix.

There are still limitations to the compiler that arise from the fact that some constructions common in lambda calculus directly require hidden automatic memory management. Chapter 3 addresses this problem with a demonstration on a (currently nonexistent) "reduced Haskell" language:

- It describes a simple solution that replaces recursive types known from functional languages by inclusion of pointed types. The solution is implemented and demonstrated in the compiler.

- It describes a possible solution to enable partial application and lazy evaluation in a language that can not store thunks in garbage-collected memory. The solution is not implemented.

- It gives several possibilities how to approach the "destructor problem" of functional languages – the problem how to reliably determine the position in a code where the object (or allocated resource) may be safely destroyed (freed) by the language, preferably in a way similar to C++ destructors. To the best of author's knowledge, no simple satisfactory solution for this problem is currently known.

Future work and open questions

Development of the compiler has shown many points where either current knowledge of the stated problem is insufficient, or the software that would provide the functionality is missing:

- More research is certainly needed on the field of low-level functional programming. Especially some solution to the destructor problem might give a way to implement a new family of interesting programming languages. There are possible constructions using linear types as defined for example by [Abr93] that may be used to solve similar problems, but those are out of scope of this thesis.

- There is lot of functionality of the TinyL compiler that was not implemented by this thesis. It is possible to extend the compiler to support compound types (i.e. user-defined structures) and a Haskell-like type system that would allow overloading or subtyping. Implementation of the proposed replacement of thunks could clearly show the benefits or applicability limits of the method.

- From practical perspective, it could also be reasonable to start from a different point and examine the possibilities of modifying some currently available compiler of functional language (for example the GHC, which shares many basic concepts with the approach chosen by this thesis [JHH+93]) to produce low-level code.

Bibliography

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical computer science*, 111(1):3–57, 1993.

[ant] Antlr parser for java. `http://www.antlr.org/`. Accessed: 2015-07-10.

[BDS13] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[bis] Bison. `http://www.gnu.org/software/bison/`. Accessed: 2015-07-10.

[Blu] Matthias Blume. Compiling with non-deterministic choice for expressing pending optimization decisions.

[bra15] Bracket pattern. `https://wiki.haskell.org/Bracket_pattern`, 2015. Accessed: 2015-07-10.

[Bru87] de NG Bruijn. Generalizing automath by means of a lambda-typed lambda calculus. *Mathematical logic and theoretical computer science/ed. by David W. Kueker, Edgar GK Lopez-Escobar, Carl H. Smith*, 106:71, 1987.

[Chu85] Alonzo Church. *The calculi of lambda-conversion*. Number 6. Princeton University Press, 1985.

[CP04] Alcino Cunha and Jorge Sousa Pinto. Point-free program transformation. 2004.

[DS97] Olivier Danvy and Ulrik P Schultz. *Lambda-dropping: transforming recursive equations into programs with block structure*, volume 32. ACM, 1997.

[Gir72] Jean-Yves Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. 1972.

[HAS10] The High Assurance Systems Programming Project HASP. *The Habit Programming Language: The Revised Preliminary Report*. Department of Computer Science, Portland State University Portland, Oregon 97207, USA, November 2010.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993.

[hhv] Hhvm php virtual machine. `http://www.hhvm.com/`. Accessed: 2015-07-10.

[HJLT05] Thomas Hallgren, Mark P Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in haskell. In *ACM SIGPLAN Notices*, volume 40, pages 116–128. ACM, 2005.

[JHA+99] S Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, et al. Haskell 98: A non-strict, purely functional language, 1999.

[JHH+93] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

[Jon00] Mark P. Jones. Typing haskell in haskell. `https://gist.github.com/chrisdone/0075a16b32bfd4f62b7b`, 2000. Accessed: 2015-07-10.

[K+36] Stephen Cole Kleene et al. Lambda-definability and recursiveness. *Duke mathematical journal*, 2(2):340–353, 1936.

[Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *ESOP'88*, pages 131–144. Springer, 1988.

[Klo80] Jan Willem Klop. Combinatory reduction systems. 1980.

[laz] lazy.rb, lazy evaluation and futures in ruby. `http://moonbase.rydia.net/software/lazy.rb/`. Accessed: 2015-07-10.

[Les11] Rebekah Leslie. *A Functional Approach to Memory-Safe Operating Systems*. Portland State University, 2011.

[LLH+10] Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. Finding error handling bugs in openssl using coccinelle. In *Dependable Computing Conference (EDCC), 2010 European*, pages 191–196. IEEE, 2010.

[Mai90] Harry G. Mairson. Deciding ml typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 382–401, New York, NY, USA, 1990. ACM.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[Nie08] Eric Niebler. Boost::proto. `http://www.boost.org/doc/libs/1_58_0/doc/html/proto.html`, 2008. Accessed: 2015-07-10.

[Pal12] Jens Palsberg. Overloading is np-complete. In *Logic and Program Semantics*, pages 204–218. Springer, 2012.

[par] Parsec parser for haskell. `https://wiki.haskell.org/Parsec`. Accessed: 2015-07-10.

[PJM02] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, 2002.

[pyp] Pypy, the fast python implementation. `http://www.pypy.org/`. Accessed: 2015-07-10.

[Rey74] John C Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[RJ07] J-W Roorda and JT Jeuring. Pure type systems for functional programming. 2007.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[Roo00] Jan-Willem Roorda. Henk 2000. `http://www.staff.science.uu.nl/~jeuri101/MSc/jwroorda/`, 2000. Accessed: 2015-07-10.

[Sit] Dorai Sitaram. Teach yourself scheme in fixnum days. `http://ds26gte.github.io/tyscheme/index-Z-H-16.html#node_chap_14`. Accessed: 2010-09-30.

[Smi94] Geoffrey S Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2):197–226, 1994.

[Tai67] William W Tait. Intensional interpretations of functionals of finite type i. *The journal of symbolic logic*, 32(02):198–212, 1967.

[Tof91] Mads Tofte. Notes on code generation using standard ml. *Lecture Notes, DIKU, Feb*, 1991.

[Tor] Linus Torvalds. Linus torvalds on c++. `http://harmful.cat-v.org/software/c++/linus`. Accessed: 2015-07-16.

[VB92] Steffen Van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

Appendix – Compiler usage

Because the compiler produced by the thesis is a simple demonstration software that lacks user documentation, following appendix is expected to be used as a simple guide for examining its properties.

A model task that will be solved by this tutorial is to produce a program that would compute the greatest common divisor of two numbers using the standard algorithm. We will show how to compile the compiler, how to write and compile the code, and how to extend the compiler with new functionality.

Compiling the compiler

The software was written and tested on a fairly standard Debian Linux and on Gentoo installations in the university lab, but should be portable to any 64-bit UNIX-like system that has following dependencies installed:

- LLVM suite version 3.5 or 3.6 (can be obtained from `http://llvm.org/releases/` or by distribution packaging method)
- GNU Make, Flex and Bison of some recent version
- A standard C library and a linker are required to run the compiled samples – the library should export at least symbols `printf` and `__isoc99_scanf` for the examples to work.[11]

Software package uses a very simple build system – all program sources are in the main directory together with a Makefile. If the dependencies are installed correctly, a single invocation of `make` should produce the compiler binary, `tlc`. The package itself is on a disc attached to the thesis, or may be downloaded from the website:

```
$ wget http://e-x-a.org/tlc/tlc.tar.gz
$ tar xzf tlc.tar.gz
$ cd tlc
$ make
...
```

[11] If the names differ on the target platform, they can be customized in source file `builtin.cpp`.

Compiling the TinyL programs

The `tlc` compiler is used very simply: It expects exactly one filename argument. The file is expected to contain the TinyL source, that will be compiled, and the result will be saved to a file with the same name as original source, only with extension `.bc`, as is the standard for LLVM bytecode.

Resulting LLVM bytecode can be translated to platform-specific assembly by a single invocation of LLVM compiler `llc`. The result is linked with the system C library that is used to execute the exported `main()` function correctly and for import of several other functions (see above), using any C linker available – the linker tested by author is GCC (versions 4.8 and 4.9).

Whole "pipeline" to compile and run a TinyL program `example.tl` would therefore be following:

```
$ tlc example.tl
$ llc example.bc
$ gcc example.s -o example
$ ./example
```

Examples that are packed with the source package are in the directory `examples/` and possess their own Makefile that executes this chain correctly.

Among the examples are:

- `rw.tl` that reads numbers from standard input and prints out some multiplies of them, until a zero is entered
- `fib.tl` that prints out first n Fibonacci numbers, for n entered on input
- `alloc.tl` that demonstrates memory allocation in a functional program
- `gcd.tl` that is the result of this tutorial

Writing the GCD program

Using a fairly standard almost-Haskell syntax, we can implement the GCD program as follows:

```
gcd a b =
  if (a<b) (gcd b a) $
  if (b==0) a (gcd (a-b) b);
```

```
main = {
  a <- read;
  b <- read;
  write (a `gcd` b);
}
```

Compiling, translating and linking:

```
$ tlc gcd.tl
$ llc gcd.bc
$ gcc gcd.s -o gcd
$ ./gcd
15504        --inputs
22236
204          --output
```

It is also possible to inspect generated assembly by viewing the `gcd.s` file, and generate a human-readable form of the `tlc` compiler output by using the LLVM disassembler `llvm-dis gcd.bc`, which generates the file `gcd.ll` with LLVM-IR source. Note especially the transformation of the tail calls and the fact that the program behaves monadically and returns the same exitcode as the function `write`, which is basically the exitcode of `printf` (that fact can be fixed by adding explicit `return 0` to the main function).

Extending the compiler

Let us improve the example. We will modify the GCD code to use the much-better modulo-division. The code for the function shall be modified as follows:

```
gcd a b =
  if (a<b) (gcd b a) $
  if (rem==0) b (gcd b rem) [rem=a%b];
```

This leaves us with an error, because the operator `%` is not implemented yet:[12]

```
error @infer.cpp:597 in `hm_leave': unbound identifier: %
```

[12]The compiler is more a research tool than actual software construction tool. Therefore, error messages sometimes contain more debugging information about the compiler than about the actual compiled program, but in most cases should be sufficient.

We must therefore add the functionality of the modulo function to the compiler. Because the builtins are virtualized, whole process is very easy. To create a new builtin, it is needed to

- decide on a builtin name (for our case, operator names are the same thing as function names – the distinction is erased in the parsing process and therefore completely invisible for most of the compiler),
- create a function that constructs the builtin type term, in our case `Uint->Uint->Uint`,
- create a function that is able to partially evaluate the term,
- create a pair of functions that direct the code generation for the term,[13]
- create a "registration" entry in the builtins table.

We will progress "backwards": In the file `builtins.cpp` we first create the builtin registration in the table, just like the other registration for binary integer operations:

```
...
  {"*", 2, type_2uint, mult, cg_stricteval, cg_mult},
  {"/", 2, type_2uint, div, cg_stricteval, cg_div},
  {"%", 2, type_2uint, mod, cg_stricteval, cg_mod},
...
```

We have actually reused some code of other operators: The `type_2uint` defines exactly the type we want, and the modulo-division requires strict evaluation of both its arguments in the entering phase of code generation.

The code for partial evaluation of binary operators on `Uint`'s is generalized by the `UINT_OP` macro, to save a lot of code duplication. In a similar manner, we only add one macro expansion:

```
...
#define divop(a,b) (a / b)
UINT_OP (div, divop)

#define modop(a,b) (a % b)
UINT_OP (mod, modop)

#define ltop(a,b) ((a<b)?1:0)
UINT_OP (cmp_lt, ltop)
...
```

[13] Detailed information about the callbacks can be found in section 3.2.5.

To generate the code, we need to determine what LLVM API builder function is used to create the modulo-division. The exact documentation for LLVM class `llvm::IRBuilder<>` can be (at the time of writing this thesis) found at `http://llvm.org/doxygen/classllvm_1_1IRBuilder.html`. From there, we get the information that the corresponding function is called `CreateURem`.

The definition of the code generation is again generalized by the `UINTOP_CG` macro that just virtualizes resulting function name and the `IRBuilder` member name for the correct operation. Therefore, we only add one more expansion:

```
...
UINTOP_CG (cg_mult, CreateMul)
UINTOP_CG (cg_div, CreateUDiv)
UINTOP_CG (cg_mod, CreateURem)

...
```

After that, the compiler should be rebuilt with `make`, and the modified GCD code will compile and work correctly. Note also that this simple expansion is already contained in the compiler package.