Friendly F# - languages and compilers.

Dr. Giuseppe Maggiore, Dr. Giulia Costantini

May 8, 2015

Contents

Chapter 1

Introduction

1.1 What is a language?

1.1.1 IMP

Expressions, variables, memory and semicolon, if, while.

Grammar Intuition of allowed/disallowed expressions based on shapes in IMP.

Type system Intuition of allowed/disallowed expressions based on shape descriptors in IMP.

Semantics Intuition of allowed/disallowed expressions based on shape descriptors in IMP.

1.1.2 λ-calculus

Abstraction, application, variables, expressions

Arithmetic

```
0 := λf x.x
1 := λf x.f x
2 := λf x.f (f x)
3 := λf x.f (f (f x))
```

We can define a successor function, which takes a number n and returns n + 1 by adding another application of f,where '(mf)x' means the function 'f' is applied 'm' times on 'x':

```
SUCC := λn f x.f (n f x)
```

Because the m-th composition of f composed with the n-th composition of f gives the m+n-th composition of f, addition can be defined as follows:

```
PLUS := λm n f x.m f (n f x)
```

`PLUS` can be thought of as a function taking two natural numbers as arguments and returning a natural number; it can be verified that

```
PLUS 2 3
```

and

```
5
```

are equivalent lambda expressions.

Logic and predicates By convention, the following two definitions (known as Church booleans) are used for the boolean values TRUE and FALSE:

```
TRUE := λx y.x
FALSE := λx y.y
```

(Note that `FALSE` is equivalent to the Church numeral zero defined above)

Then, with these two λ-terms, we can define some logic operators (these are just possible formulations; other expressions are equally correct):

```
AND := λp q.p q p
OR := λp q.p p q
NOT := λp a b.p b a
IFTHENELSE := λp a b.p a b
```

A predicate is a function that returns a boolean value. The most fundamental predicate is ISZERO, which returns TRUE if its argument is the Church numeral 0, and FALSE if its argument is any other Church numeral:

```
ISZERO := λn.n (λx.FALSE) TRUE
```

Chapter 2

Syntax

2.1 Formal grammars

Allowed and disallowed shapes, formally defined.

A formal grammar consists of a finite set of production rules (left-hand side $\rightarrow$ right-hand side), where each side consists of a sequence of the following symbols:

- a finite set of *nonterminal symbols* (indicating that some production rule can yet be applied)
- a finite set of *terminal symbols* (indicating that no production rule can be applied)
- a *start symbol* (a distinguished nonterminal symbol)

Nonterminals are usually denoted by uppercase letters, terminals by lowercase letters, and the start symbol by S.

A formal grammar *defines* (or *generates*) a formal language.

A formal language is a (usually infinite) set of finite-length sequences of symbols (i.e. strings) that may be constructed by applying production rules starting from just the start symbol. A rule may be applied to a sequence of symbols by replacing an occurrence of the symbols on the left-hand side of the rule with those that appear on the right-hand side. A sequence of rule applications is called a *derivation.*

Such a grammar defines the formal language: all words consisting solely of terminal symbols which can be reached by a derivation from the start symbol.

Example Formal grammar for IMP.

Formal grammar for the lambda-calculus.

Example derivations.

2.2 The hierarchy

Not all languages are created equal.

Formal grammars are classified in the so-called *Chomsky hierarchy* w.r.t. their expressive power.

It is a hierarchy in the sense that each level contains everything expressed at the lower levels.

Type-0 grammars (unrestricted grammars) include all formal grammars, and generate exactly all languages that can be recognized by a Turing machine.

Example: given a programming language semantics, a program, and an output, determine if the output can be reached through the semantics from the original program.

Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with A a nonterminal and α, β and γ strings of terminals and/or nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (LBA). An LBA is any non-deterministic (multiple rules at the same time) Turing machine whose tape is bounded by a constant times the length of the input.

One of the simplest context-sensitive, but not context-free languages is $L = \{a^n b^n c^n : n \geq 1\}$: the language of all strings consisting of n occurrences of the symbol a, then n b's, then n c's (abc, $aabbcc$, $aaabbbccc$, etc.).

1. $S \rightarrow abc$
2. $S \rightarrow aSBc$
3. $cB \rightarrow WB$
4. $WB \rightarrow WX$
5. $WX \rightarrow BX$
6. $BX \rightarrow Bc$
7. $bB \rightarrow bb$

Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and/or nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton (NDPA). An NDPA is a recursive procedure that uses the stack to add multiple sub-problems to the working stack. The subset of *deterministic context-free languages* is the theoretical basis for the phrase structure of most programming languages, though their syntax also includes context-sensitive name resolution due to declarations and scope.

A common canonical example is two different kinds of matching nested parentheses, described by the productions:

1. $S \to SS$
2. $S \to ()$
3. $S \to (S)$
4. $S \to []$
5. $S \to [S]$

with terminal symbols []() and nonterminal S.

The following sequence can be derived in that grammar:

$$([[[()()[][]]]([])])$$

Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:

1. $S \to x$
2. $S \to y$
3. $S \to z$
4. $S \to S + S$
5. $S \to S - S$
6. $S \to S * S$
7. $S \to S/S$
8. $S \to (S)$

This grammar can, for example, generate the string

$$(x + y) * x - z * y/(x + x)$$

Type-3 grammars (regular grammars) generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal (right regular). Alternatively, the right-hand side of the grammar can consist of a single terminal, possibly preceded by a single nonterminal (left regular); these generate the same languages – however, if left-regular rules and right-regular rules are combined, the language need no longer be regular. The rule $S \rightarrow \epsilon$ is also allowed here if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

An example of a (right) regular grammar consists of the following rules:

1. $S \rightarrow aS$
2. $S \rightarrow bA$
3. $A \rightarrow \epsilon$
4. $A \rightarrow cA$

This grammar describes the same language as the regular expression a^*bc^*.

Chapter 3

Semantics

Operational semantics are a category of formal programming language semantics in which certain desired properties of a program, such as correctness, safety or security, are verified by constructing proofs from logical statements about its execution and procedures, rather than by attaching mathematical meanings to its terms (denotational semantics). Operational semantics are classified in two categories: structural operational semantics (or small-step semantics) formally describe how the individual steps of a computation take place in a computer-based system. By opposition natural semantics (or big-step semantics) describe how the overall results of the executions are obtained. Other approaches to providing a formal semantics of programming languages include axiomatic semantics and denotational semantics.

The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. These sequences then are the meaning of the program. In the context of functional programs, the final step in a terminating sequence returns the value of the program. (In general there can be many return values for a single program, because the program could be nondeterministic, and even for a deterministic program there can be many computation sequences since the semantics may not specify exactly what sequence of operations arrives at that value.)

3.1 (Structural) operational semantics

Structural operational semantics (also called structured operational semantics or small-step semantics) was introduced by Gordon Plotkin in (Plotkin81) as a logical means to define operational semantics. The basic idea behind SOS is to define the behavior of a program in terms of the behavior of its parts, thus providing a structural, i.e., syntax oriented and inductive, view on operational semantics. An SOS specification defines the behavior of a program in terms of a (set of) transition relation(s). SOS specifications take the form of a set of inference rules that define the valid transitions of a composite piece of syntax in terms of the transitions of its components.

For a simple example, we consider part of the semantics of a simple programming language; proper illustrations are given in Plotkin81 and Hennessy90, and other textbooks. Let C_1, C_2 range over programs of the language, and let s range over states (e.g. functions from memory locations to values). If we have expressions (ranged over by E), values (V) and locations (L), then a memory update command would have semantics:

$$\frac{\langle E, s\rangle \Rightarrow V}{\langle L := E\ ,\ s\rangle \longrightarrow (s \uplus (L \mapsto V))}$$

Informally, the rule says that "if the expression E in state s reduces to value V, then the program L:=E will update the state s with the assignment L=V".

The semantics of sequencing can be given by the following three rules:

$$\frac{\langle C_1, s\rangle \longrightarrow s'}{\langle C_1; C_2\ , s\rangle \longrightarrow \langle C_2, s'\rangle} \qquad \frac{\langle C_1, s\rangle \longrightarrow \langle C_1', s'\rangle}{\langle C_1; C_2\ , s\rangle \longrightarrow \langle C_1'; C_2\ , s'\rangle} \qquad \frac{}{\langle \mathbf{skip}, s\rangle \longrightarrow s}$$

Informally, the first rule says that, if program C_1 in state s finishes in state s', then the program $C_1; C_2$ in state s will reduce to the program C_2 in state s'. (You can think of this as formalizing "You can run C_1, and then run C_2 using the resulting memory store.) The second rule says that if the program C_1 in state s can reduce to the program C_1' with state s', then the program $C_1; C_2$ in state s will reduce to the program $C_1'; C_2$ in state s'. (You can think of this as formalizing the principle for an optimizing compiler: "You are allowed to transform C_1 as if it were stand-alone, even if it is just the first part of a program.") The semantics is structural, because the meaning of the sequential program $C_1; C_2$, is defined by the meaning of C_1 and the meaning of C_2.

If we also have Boolean expressions over the state, ranged over by B, then we can define the semantics of the while command:

$$\frac{\langle B, s\rangle \Rightarrow \mathbf{true}}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \longrightarrow \langle C; \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle} \qquad \frac{\langle B, s\rangle \Rightarrow \mathbf{false}}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \longrightarrow s}$$

Such a definition allows formal analysis of the behavior of programs, permitting the study of relations between programs. Important relations include simulation preorders and bisimulation. These are especially useful in the context of concurrency theory.

Thanks to its intuitive look and easy to follow structure, SOS has gained great popularity and has become a de facto standard in defining operational semantics. As a sign of success, the original report (so-called Aarhus report) on SOS (Plotkin81) has attracted more than 1000 citations according to the CiteSeer [1], making it one of the most cited technical reports in Computer Science.

3.2 Reduction semantics

Reduction semantics are an alternative presentation of operational semantics using so-called reduction contexts. The method was introduced by Robert Hieb and Matthias Felleisen in 1992 as a technique for formalizing an equational theory for control and state. For example, the grammar of a simple call-by-value lambda calculus and its contexts can be given as:

$$e = v \mid (e\ e) \mid x \qquad v = \lambda x.e \qquad C = [\,] \mid (C\ e) \mid (v\ C)$$

The contexts C include a hole [] where a term can be plugged in. The shape of the contexts indicate where reduction can occur (i.e., a term can be plugged into) a term. To describe a semantics for this language, axioms or reduction rules are provided:

$$(\lambda x.e\ v) \longrightarrow e\ [x/v] \quad (\beta)$$

This single axiom is the beta rule from the lambda calculus. The reduction contexts show how this rule composes with more complicated terms. In particular, this rule can trigger for the argument position of an application like $((\lambda x.x\ \lambda x.x)\lambda x.(x\ x))$ because there is a context $([\,]\ \lambda x.(x\ x))$ that matches the term. In this case, the contexts uniquely decompose terms so that only one reduction is possible at any given step. Extending the axiom to match the reduction contexts gives the compatible closure. Taking the reflexive, transitive closure of this relation gives the reduction relation for this language.

The technique is useful for the ease in which reduction contexts can model state or control constructs (e.g., continuations). In addition, reduction semantics have been used to model object-oriented languages,[2] contract systems, and other language features.

Chapter 4

Type systems

In programming languages, a type system is a collection of rules that assign a property called a type to the various constructs (such as variables, expressions, functions or modules) that a computer program is composed of.[1] The main purpose of a type system is to reduce bugs in computer programs[2] by defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way.

This checking can happen statically (at compile time), dynamically (at run time), or it can happen as a combination of static and dynamic checking. Type systems have other purposes as well, such as enabling certain compiler optimizations, allowing for multiple dispatch, providing a form of documentation, etc.

A type-system associates a type with each computed value. By examining the flow of these values, a type system attempts to ensure or prove that no type errors can occur. The particular type system in question determines exactly what constitutes a type error, but in general the aim is to prevent operations expecting a certain kind of value from being used with values for which that operation does not make sense (logic errors); memory errors will also be prevented. Type systems are often specified as part of programming languages, and built into the interpreters and compilers for them; although the type system of a language can be extended by optional tools that perform additional kinds of checks using the language's original type syntax and grammar.

4.1 Fundamentals

Formally, type theory studies type systems. A programming language must have occurrence to type check using the type system whether at compiler time or runtime, manually annotated or automatically inferred. As Mark Manasse concisely put it:[3]

The fundamental problem addressed by a type theory is to ensure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension.

Assigning a data type (*typing*) meaning to a sequences of bits such as a value in memory or some object such as a variable.

The hardware of a general purpose computer is unable to discriminate between for example a memory address and an instruction code, or between a character, an integer, or a floating-point number, because it makes no intrinsic distinction between any of the possible values that a sequence of bits might mean.[note 1] Associating a sequence of bits with a type conveys that meaning to the programmable hardware to form a symbolic system composed of that hardware and some program.

A program associates each value with at least one particular type, but it also can occur that one value is associated with many subtypes. Other entities, such as objects, modules, communication channels, dependencies can become associated with a type. Even a type can become associated with a type. An implementation of some type system could in theory associate some identifications named this way: data type – a type of a value class – a type of an object kind – a type of a type, or metatype

These are the abstractions that typing can go through, on a hierarchy of levels contained in a system.

When a programming language evolves a more elaborate type system, it gains a more finely grained rule set than basic type checking, but this comes at a price when the type inferences (and other properties) become undecidable, and when more attention must be paid by the programmer to annotate code or to consider computer-related operations and functioning. It is challenging to find a sufficiently expressive type system that satisfies all programming practices in a type safe manner.

The more type restrictions that are imposed by the compiler, the more strongly typed a programming language is. Strongly typed languages often require the programmer to make explicit conversions in contexts where an implicit conversion would cause no harm. Pascal's type system has been described as "too strong" because, for example, the size of an array or string is part of its type, making some programming tasks difficult.[4][5] Haskell is also strongly typed but its types are automatically inferred so that explicit conversions are unnecessary.

A programming language compiler can also implement a dependent type or an effect system, which enables even more program specifications to be verified by a type checker. Beyond simple value-type pairs, a virtual "region" of code is associated with an "effect" component describing what is being done with what, and enabling for example to "throw" an error report. Thus the symbolic system may be a type and effect system, which endows it with more safety checking than type checking alone.

Whether automated by the compiler or specified by a programmer, a type system makes program behavior illegal that is outside the type-system rules. Advantages provided by programmer-specified type systems include: Abstraction (or modularity) – Types enable programmers to think at a higher level than the bit or byte, not bothering with low-level implementation. For example, programmers can begin to think of a string as a collection of character values instead of as a mere array of bytes. Higher still, types enable programmers to think about and express interfaces between two of any-sized subsystems. This enables more levels of localization so that the definitions required for interoperability of the subsystems remain consistent when those two subsystems communicate. Documentation – In more expressive type systems, types can serve as a form of documentation clarifying the intent of the programmer. For instance, if a programmer declares a function as returning a timestamp type, this documents the function when the timestamp type can be explicitly declared deeper in the code to be integer type.

Advantages provided by compiler-specified type systems include: Optimization – Static type-checking may provide useful compile-time information. For example, if a type requires that a value must align in memory at a multiple of four bytes, the compiler may be able to use more efficient machine instructions. Safety – A type system enables the compiler to detect meaningless or probably invalid code. For example, we can identify an expression 3 / "Hello, World" as invalid, when the rules do not specify how to divide an integer by a string. Strong typing offers more safety, but cannot guarantee complete type safety.

Type safety contributes to program correctness, but can only guarantee correctness at the expense of making the type checking itself an undecidable problem[citation needed]. In a type system with automated type checking a program may prove to run incorrectly yet be safely typed, and produce no compiler errors. Division by zero is an unsafe and incorrect operation, but a type checker running only at compile time doesn't scan for division by zero in most programming languages, and then it is left as a runtime error. To prove the absence of these more-general-than-types defects, other kinds of formal methods, collectively known as program analyses, are in common use. Alternatively, a sufficiently expressive type system, such as in dependently typed languages, can prevent these kinds of errors (for example, expressing "the type of non-zero numbers"). In addition software testing is an empirical method for finding errors that the type checker cannot detect.

4.2 Type checking

The process of verifying and enforcing the constraints of types – type checking – may occur either at compile-time (a static check) or runtime (a dynamic check). If a language specification requires its typing rules strongly (i.e., more or less allowing only those automatic type conversions that do not lose information), one can refer to the process as strongly typed, if not, as weakly typed. The terms are not usually used in a strict sense.

4.2.1 Static type-checking

Static type-checking is the process of verifying the type safety of a program based on analysis of a program's text (source code). If a program passes a static type-checker, then the program is guaranteed to satisfy some set of type-safety properties for all possible inputs.

Because static type-checking operates on a program's text, it allows many bugs to be caught early in the development cycle.

Static type-checking can be thought of as a limited form of program verification (see type safety). In a type-safe language, static type-checking can also be thought of as an optimization. If a compiler can prove that a program is well-typed, then it does not need to emit dynamic safety checks, allowing the resulting compiled binary to run faster.

Static type-checking for Turing-complete languages is inherently conservative. That is, if a type system is both sound (meaning that it rejects all incorrect programs) and decidable (meaning that it is possible to write an algorithm which determines whether a program is well-typed), then it will always be possible to define a program which is well-typed but which does not satisfy the type-checker.[6] For example, consider a program containing the code:

if ¡complex test¿ then ¡do something¿ else ¡generate type error¿

Even if the expression ¡complex test¿ always evaluates to true at run-time, most type-checkers will reject the program as ill-typed, because it is difficult (if not impossible) for a static analyzer to determine that the else branch will not be taken.[7] Conversely, a static type-checker will quickly detect type errors in rarely used code paths. Without static type checking, even code coverage tests with 100

A number of useful and common programming language features cannot be checked statically, such as downcasting. Therefore, many languages will have both static and dynamic type-checking; the static type-checker verifies what it can, and dynamic checks verify the rest.

Many languages with static type-checking provide a way to bypass the type checker. Some languages allow programmers to choose between static and dynamic type safety. For example, C# distinguishes between "statically-typed" and "dynamically-typed" variables; uses of the former are checked statically, while uses of the latter are checked dynamically. Other languages allow users to write code which is not type-safe. For example, in C, programmers can freely cast a value between any two types which have the same size.

For a list of languages with static type-checking, see the category for statically typed languages.

4.2.2 Dynamic type-checking and runtime type information

Dynamic type-checking is the process of verifying the type safety of a program at runtime. Implementations of dynamically type-checked languages generally associate each runtime object with a "type tag" (i.e., a reference to a type) containing its type information. This runtime type information (RTTI) can also be used to implement dynamic dispatch, late binding, downcasting, reflection, and similar features.

Most type-safe languages include some form of dynamic type-checking, even if they also have a static type checker. The reason for this is that many useful features or properties are difficult or impossible to verify statically. For example, suppose that a program defines two types, A and B, where B is a subtype of A. If the program tries to convert a value of type A to type B, which is known as downcasting, then the operation is legal only if the value being converted is actually a value of type B. Therefore, a dynamic check is needed to verify that the operation is safe; this requirement is one of the criticisms of downcasting.

By definition, dynamic type-checking may cause a program to fail at runtime. In some programming languages, it is possible to anticipate and recover from these failures. In others, type-checking errors are considered fatal.

Programming languages which include dynamic type-checking but not static type-checking are often called "dynamically-typed programming languages". For a list of such languages, see the category for dynamically typed programming languages.

4.2.3 "Strong" and "weak" type systems

Main article: Strong and weak typing

Languages are often colloquially referred to as "strongly typed" or "weakly typed". In fact, there is no universally accepted definition of what these terms mean. In general, there are more precise terms to represent the differences between type systems that lead people to call them "strong" or "weak".

In computer programming, programming languages are often colloquially referred to as strongly typed or weakly typed. These terms do not have a precise definition, but in general a strongly typed language is more likely to generate an error or refuse to compile if the argument passed to a function does not closely match the expected type. On the other hand, a very weakly typed language may produce unpredictable results or may perform implicit type conversion.[1]

4.2.4 Type safety and memory safety

A third way of categorizing the type system of a programming language uses the safety of typed operations and conversions. Computer scientists consider a language "type-safe" if it does not allow operations or conversions that violate the rules of the type system.

Some observers use the term memory-safe language (or just safe language) to describe languages that do not allow programs to access memory that has not been assigned for their use. For example, a memory-safe language will check array bounds, or else statically guarantee (i.e., at compile time before execution) that array accesses out of the array boundaries will cause compile-time and perhaps runtime errors.

4.3 Type Inference

Type inference refers to the automatic deduction of the data type of an expression in a programming language. If some, but not all, type annotations are already present it is referred to as type reconstruction. The opposite operation of type inference is called type erasure.

It is a feature present in some strongly statically typed languages. It is often characteristic of, but not limited to, functional programming languages in general. Some languages that include type inference are

ML, OCaml, F#, Haskell, Scala, D, Clean, Opa, Rust, Swift, Visual Basic (starting with version 9.0), C# (starting with version 3.0) and C++11. The ability to infer types automatically makes many programming tasks easier, leaving the programmer free to omit type annotations while still permitting type checking.

Type inference is the ability to automatically deduce, either partially or fully, the type of an expression at compile time. The compiler is often able to infer the type of a variable or the type signature of a function, without explicit type annotations having been given. In many cases, it is possible to omit type annotations from a program completely if the type inference system is robust enough, or the program or language is simple enough.

To obtain the information required to infer the type of an expression, the compiler either gathers this information as an aggregate and subsequent reduction of the type annotations given for its subexpressions, or through an implicit understanding of the type of various atomic values (e.g. true : Bool; 42 : Integer; 3.14159 : Real; etc.). It is through recognition of the eventual reduction of expressions to implicitly typed atomic values that the compiler for a type inferring language is able to compile a program completely without type annotations.

In the case of complex forms of higher-order programming and polymorphism, it is not always possible for the compiler to infer as much, however, and type annotations are occasionally necessary for disambiguation. For instance, type inference with polymorphic recursion is known to be undecidable. Furthermore, explicit type annotations can be used to optimize code by forcing the compiler to use a more specific (faster/smaller) type than it had inferred.[1]

From a program analysis point of view, type inference is a special case of points-to analysis that uses a type abstraction on pointer targets.

4.3.1 Hindley-Milner algorithm

In type theory and functional programming, Hindley–Milner (HM) (also known as Damas–Milner or Damas–Hindley–Milner) is a classical type system for the lambda calculus with parametric polymorphism, first described by J. Roger Hindley[1] and later rediscovered by Robin Milner.[2] Luis Damas contributed a close formal analysis and proof of the method in his PhD thesis.[3][4]

Among HM's more notable properties is completeness and its ability to deduce the most general type of a given program without the need of any type annotations or other hints supplied by the programmer. Algorithm W is a fast algorithm, performing type inference in almost linear time with respect to the size of the source, making it practically usable to type large programs.[note 1] HM is preferably used for functional languages.

Algorithm W

$$\frac{x : \sigma \in \Gamma \quad \tau = \mathit{inst}(\sigma)}{\Gamma \vdash x : \tau} \qquad [\mathtt{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \tau' = \mathit{newvar} \quad \mathit{unify}(\tau_0,\ \tau_1 \rightarrow \tau')}{\Gamma \vdash e_0\ e_1 : \tau'} \qquad [\mathtt{App}]$$

$$\frac{\tau = \mathit{newvar} \quad \Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x\ .\ e : \tau \rightarrow \tau'} \qquad [\mathtt{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma,\ x : \bar{\Gamma}(\tau) \vdash e_1 : \tau'}{\Gamma \vdash \mathtt{let}\ x = e_0\ \mathtt{in}\ e_1 : \tau'} \qquad [\mathtt{Let}]$$

Chapter 5

Conclusions