Systematic Compiler Construction

Michael Sperber

November 28, 2001

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme der folgenden sind vorbehalten:

Studenten der Informatik dürfen diese Dokument für ihre persönlichen Lehrzwecke verwenden.

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken, Dateikopien oder Paperkopien. Insbesondere darf dieses Skriptum nicht in irgendeiner Weise vetrieben werden und es dürfen keine Kopien der Dateien auf öffentlich zugänglichen Servern angelegt werden.

Copyright © Michael Sperber, 1999

Chapter 3

The Lambda Calculus

The abstract syntax for Mini-Caml presented in the previous chapter is already fairly compact. However, the real meaning of most of the constructs there is largely unclear. The Caml manual offers an informal description of what each construct does. Unfortunately, this is not sufficient for building a real implementation. In the compiler writing business, it is obviously necessary to have a precise idea of the meaning of each construct in the language. Therefore, after having attacked Mini-Caml from the front, that is, its appearance to the user, it is now necessary to examine the formal system that forms the basis of Caml, and ideed most other programming languages: the lambda calculus. It will essentially serve as an intermediate language into which the compiler will translate Mini-Caml programs and from which it will generate machine code.

3.1 Syntax and reduction semantics

The lambda calculus is a logical reduction calculus: It consists of a language for terms and some reduction rules which describe how to transform terms into other terms.

3.1 Definition (Language of the lambda calculus)
Let ⟨var⟩ stand for an element of a countable set of *variables* V. The set E of *lambda terms* consists of terms described the following grammar:

$$\langle\text{exp}\rangle \longrightarrow \langle\text{var}\rangle \mid (\langle\text{exp}\rangle \; \langle\text{exp}\rangle) \mid (\lambda \; \langle\text{var}\rangle . \langle\text{exp}\rangle)$$

Terms of the form $(e_0\,e_1)$ are *applications*, terms of the form $\lambda x.e$ are *abstractions* with *body* e.

To save on redundant parentheses, the following conventions apply to the representation of lambda calculus terms:

- Applications are left-associative.
- The body of an abstraction reaches as far to the right as possible.
- $\lambda xy.e$ stands for $\lambda x.\lambda y.e$ (analogously for more arguments).

□

Intuitively, the objects of the lambda calculus are functions: An abstraction denotes a function, an application an—application. However, there are different methods for

evaluating terms containing functions: first inner terms, then outer, or vice versa, left-to-right, or right-to-left. The lambda calculus, in its original form, does not prescribe a particular evaluation strategy—it generalizes over all of them. Later, suitable restrictions will describe such strategies and become more similar to actual programming languages. To properly understand the implications of committing to a particular strategy, it is necessary to first examine the general theory, however.

A description of the meaning of lambda terms requires some auxiliary definitions. The letter e usually stands for an arbitrary lambda term, v for a variable:

3.2 Definition (Free and bound variables)
The functions free, bound : $E \to \mathcal{P}(V)$ return the set of *free* or *bound* variables of a lambda term, respectively.

$$\begin{aligned}
\text{free}(v) &:= \{v\} \\
\text{free}(e_0\ e_1) &:= \text{free}(e_0) \cup \text{free}(e_1) \\
\text{free}(\lambda v.e) &:= \text{free}(e) \setminus \{v\}
\end{aligned}$$

$$\begin{aligned}
\text{bound}(v) &:= \varnothing \\
\text{bound}(e_0\ e_1) &:= \text{bound}(e_0) \cup \text{bound}(e_1) \\
\text{bound}(\lambda v.e) &:= \text{bound}(e) \cup \{v\}
\end{aligned}$$

Furthermore, $\text{var}(e) := \text{free}(e) \cup \text{bound}(e)$ is the *set of variables* of e. A lambda term e is *closed* (e is a *combinator*) iff $\text{free}(e) = \varnothing$.

□

3.3 Definition (Substitution)
For $e, f \in E$, $e[v \mapsto f]$ is inductively defined by:

$$\begin{aligned}
v[v \mapsto f] &:= f \\
x[v \mapsto f] &:= x && x \neq v \\
(\lambda v.e)[v \mapsto f] &:= \lambda v.e \\
(\lambda x.e)[v \mapsto f] &:= \lambda x.(e[v \mapsto f]) && x \neq v, x \notin \text{free}(f) \\
(\lambda x.e)[v \mapsto f] &:= \lambda x'.(e[x \mapsto x'][v \to f]) && x \neq v, x \in \text{free}(f), x' \notin \text{free}(e) \cup \text{free}(f) \\
(e_0\ e_1)[v \mapsto f] &:= (e_0[v \mapsto f])(e_1[v \mapsto f])
\end{aligned}$$

□

3.4 Definition (Reduction rules)
There are three different reduction rules for the lambda calculus: α reduction, β reduction, and η reduction. Each is a binary relation on lambda terms.

$$\begin{aligned}
\lambda x.e &\to_\alpha \lambda y.e[x \mapsto y] && y \notin \text{free}(e) \\
(\lambda v.e)\ f &\to_\beta e[v \mapsto f] \\
(\lambda x.e\ x) &\to_\eta e && x \notin \text{free}(e)
\end{aligned}$$

Each reduction rule is compatibly extended to also work on subterms.

For $x \in \{\alpha, \beta, \gamma\}$, $\overset{*}{\to}_x$ is the reflexive-transitive closure, and $\leftrightarrow_x$ is its symmetric closure, and $\overset{*}{\leftrightarrow}_x$ is its reflexive-transitive-symmetric closure.

□

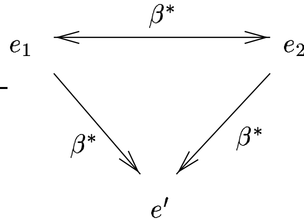
Note that β-reduction corresponds closely to the intuitive notion of function application.

3.5 Definition (Normal form)
Let e be a lambda term. A lambda term e' is a *normal form* of e iff $e \xrightarrow{*}_\beta e'$ and if there is no e'' with $e' \rightarrow_\beta e''$.

Lambda terms with equivalent (equal module α reduction) normal forms exhibit the same behavior. The reverse is not always true. Also, some lambda terms do not have a normal form:

$$(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow_\beta (\lambda x.x\ x)(\lambda x.x\ x)$$

3.6 Theorem (Church-Rosser)
The β reduction has the *Church-Rosser property*:



In words: For all lambda terms e_1, e_2 with $e_1 \overset{*}{\leftrightarrow}_\beta e_2$, there is a lambda term e' with $e_1 \xrightarrow{*}_\beta e'$ and $e_2 \xrightarrow{*}_\beta e'$.

□

3.7 Corollary (Corollary)
A lambda term e has at most one normal form modulo α reduction.

□

3.2 The lambda calculus as a programming language

The lambda calculus may at first seem a fairly silly way to go about describing functions: In the world of the lambda calculus, there is nothing *but* functions, and the scarcity of its language seem to allow for only the most primitive computations (if any). Nevertheless, the lambda calculus has the same power as any programming language. (The theoreticians say it is "Turing-equivalent.")

Adding conventional programming language constructs to the lambda calculus is somewhat tedious (if not particularly) hard. As such, they are not directly usable for programming language implementations. However, it is good to have a working knowledge of the necessary mechanisms if only to get some practice dealing with the calculus. In practice, the "pure" lambda calculus gives way to an "applied" lambda calculus which has the necessary built-in primitive operations to directly perform useful computations.

What constructs are necessary for useful computations? The lambda calculus at first glance seems to lack the following fundamental ingredients:

- some sort of conditional and booleans,
- numbers, and
- recursion.

While a typical applied lambda calculus typically has all of these, it is possible to model all of them in the pure one:

Conditionals in functional languages usually have the form `if` e `then` e_1 `else` e_2: Depending on the (boolean) result of evaluating e, the conditional either "selects" e_1 or e_2. The way to go in the lambda calculus is to give booleans themselves an "active" interpretation an make them actually *perform* the selection. Thus, *true* is a lambda term that selects the first of two arguments, and *false* is one that selects the second:

$$\begin{aligned} \mathit{true} &:= \lambda xy.x \\ \mathit{false} &:= \lambda xy.y \end{aligned}$$

Consequently, the conditional itself degenerates to simple application:

$$\mathit{if} := \lambda txy.t\ x\ y$$

It is straightforward to verify that *if* actually adheres to the intuition. For a true test, the beta reduction goes like this:

$$\begin{aligned} \mathit{if}\ \mathit{true}\ e_1\ e_2 &= (\lambda txy.t\ x\ y)\ \mathit{true}\ e_1\ e_2 \\ &\rightarrow_\beta (\lambda xy.\mathit{true}\ x\ y)\ e_1\ e_2 \\ &\rightarrow_\beta^2 \mathit{true}\ e_1\ e_2 \\ &= (\lambda xy.x)\ e_1\ e_2 \\ &\rightarrow_\beta (\lambda y.e_1)\ e_2 \\ &\rightarrow_\beta e_1 \end{aligned}$$

For *false*, the proof goes analogously.

Numbers are slightly more involved. Several methods exist to represent numbers by lambda terms. One is to use *Church numerals*. The Church numeral $\lceil n \rceil$ of some natural number n is a function which performs an n-fold application. (Hence, $\lceil 0 \rceil$ is the identity.)

$$\lceil n \rceil := \lambda f \lambda x.f^n(x)$$

where

$$f^n(e) := \begin{cases} e & \text{if } n = 0 \\ f(f^{n-1}(e)) & \text{otherwise} \end{cases}$$

The successor function adds an application:

$$\mathit{succ} := \lambda n.\lambda f \lambda x.n\ f\ (f\ x)$$

The predecessor is somewhat more complicated:

$$\mathit{pred} := \lambda x.\lambda y.\lambda z.x\ (\lambda p.\lambda q.q\ (p\ y))\ ((\lambda x.\lambda y.x)\ z)\ (\lambda x.x)$$

(A proof that it actually does subtract one from a Church numeral is a worthwhile exercise.)

Also, a test for zero is possible:

$$\mathit{zerop} := \lambda n.n\ (\lambda x.\mathit{false})\ \mathit{true}$$

Again, a simple test case serves as an example:

$$\begin{aligned} \mathit{zerop}\ \lceil 0 \rceil &= (\lambda n.n\ (\lambda x.\mathit{false})\ \mathit{true})\ \lceil 0 \rceil \\ &\rightarrow_\beta \lceil 0 \rceil\ (\lambda x.\mathit{false})\ \mathit{true} \\ &= (\lambda f.\lambda x.x)\ (\lambda x.\mathit{false})\ \mathit{true} \\ &\rightarrow_\beta (\lambda x.x)\ \mathit{true} \\ &\rightarrow_\beta \mathit{true} \end{aligned}$$

The only thing missing now is recursion. Since a recursive function needs to refer to itself, it needs to receive a name which is passed to it by a magical term called a *fixpoint combinator*. The magic is sufficient to warrant a theorem:

3.8 Theorem (Fixpoint theorem)
For every lambda term F there is a lambda term X with $F\ X \overset{*}{\leftrightarrow}_\beta X$.

Proof:

Choose $X := Y\ F$ with

$$Y := \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)).$$

Then:

$$\begin{aligned} Y\ F &= (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))\ F \\ &\rightarrow_\beta (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x)) \\ &\rightarrow_\beta F\ ((\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x))) \\ &\leftarrow_\beta F\ ((\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ F) \\ &= F\ (Y\ F) \end{aligned}$$

□

A fixpoint combinator suitable for multiple recursion involves no new principles, but is very nasty to formulate.

3.3 Evaluation strategies

Since vanilla β reduction applies to arbitrary subterms, having normal forms is of limited value: It is not clear how to compute them because success is highly dependent on the order in which subterms are subject to reduction. In the practice of programming, full normal forms are rarely important. Instead, it is usually sufficient to evaluate lambda terms to the point where they are simple values or abstractions; it is not necessary to evaluate anything "‘inside the lambda."’ This leads to the notion of *weak head-normal forms*:

3.9 Definition (Weak head-normal form)
A lambda term which is an abstraction is called a *value* or a *weak head-normal form*. All other lambda terms are called *expression juxtapositions*.

□

Next, it is desirable to formulate deterministic strategies that prescribe how to evaluate a λ term to its weak head-normal form, so-called *evaluation strategies*. A succinct formalism for describing such strategies is the concept of an evaluation context:

3.10 Definition (Evaluation contexts)
An *evaluation context* C is a mapping $E \rightarrow E$ denoted by a lambda term with a "hole," i.e. one of its subterms is the literal [], and the mapping works by substituting its argument for the hole. The notation for substituting a lambda term e for such a hole is $C[e]$. Thus, an evaluation context has one of the following forms:

$$\begin{array}{rcll} C[e] & := & e & \text{where } e \in E \\ & & \lambda x.C'[e] & C' \text{ is an evaluation context} \\ & & (C'[e]\ e') & C' \text{ is an evaluation context} \\ & & (e'\ C'[e]) & C' \text{ is an evaluation context} \end{array}$$

□

The two important strategies for computing weak head-normal forms are called *call-by-name* and *call-by-value*:

3.11 Definition (Call-by-name lambda calculus)
The relation $\rightarrow_{\beta n}$—the *call-by-name standard reduction*—is defined by:

$$(\lambda x.e)\ f \rightarrow_{\beta n} e[x \mapsto f]$$

extended to subterms which are in an evaluation context as follows:

$$E_n ::= [\,] \mid E_n\ e.$$

□

The call-by-name lambda calculus may evaluate subexpression multiple times. Therefore, most real-world programming language use a different strategy derived from a leftmost-innermost strategy:

3.12 Definition (Call-by-value lambda calculus)
Let w always stand for a value, and e for an expression juxtaposition. The relation $\rightarrow_{\beta v}$ (the *call-by-value standard reduction*) on lambda terms is defined by

$$(\lambda x.e)\ w \rightarrow_{\beta v} e[x \mapsto w].$$

$\rightarrow_{\beta v}$ is extended to work on subterms which are in an evaluation context as follows:

$$E_v ::= [\,] \mid (w\ E_v) \mid (E_v\ e).$$

□

3.4 Recursive applicative program schemes

Neither Church numerals nor the fixpoint combinator are particularly efficient ways of implementing realistic programs. Therefore, applications in the real world usually use the lambda calculus in some applied form which already contains essential primitive data types and recursion in a primitive form. For good measure, applied lambda calculi usually also contain booleans and a conditional. One particular way of formulating an applied lambda calculus is to use *recursive applicative program schemes*. Such a program is essentially a set of equations, the right-hand sides of which are lambda terms.

3.13 Definition (Recursive applicative program schemes)
Let C be a countable set of names for *constants* including *true* and *false* with $V \cap C = \varnothing$, and let $\langle\text{const}\rangle$ be a grammar symbol that stands for such an operator. Let F be a countable set of *function names* with $F \cap V = \varnothing$ and $V \cap C = \varnothing$, and let $\langle\text{name}\rangle$ be a grammar symbol that stands for such a name. An *applied lambda term* is a term adhering to the following syntax:

$$\begin{aligned}
\langle\text{exp}\rangle \longrightarrow\ & \langle\text{var}\rangle \\
\mid\ & \langle\text{name}\rangle \\
\mid\ & \langle\text{const}\rangle \\
\mid\ & (\ \lambda\ \langle\text{var}\rangle\,.\,\langle\text{exp}\rangle\ ) \\
\mid\ & (\ \langle\text{exp}\rangle\ \langle\text{exp}\rangle\ ) \\
\mid\ & \textit{let}\ \langle\text{var}\rangle\ =\ \langle\text{exp}\rangle\ \textit{in}\ \langle\text{exp}\rangle \\
\mid\ & \textit{if}\ \langle\text{exp}\rangle\ \textit{then}\ \langle\text{exp}\rangle\ \textit{else}\ \langle\text{exp}\rangle
\end{aligned}$$

The applied lambda terms form a set E'.

A *recursive applicative program scheme* is a set of equations together with a lambda term with the following form:

$$\langle\text{scheme}\rangle \longrightarrow \langle\text{equation}\rangle^* \ \langle\text{exp}\rangle$$
$$\langle\text{equation}\rangle \longrightarrow \langle\text{name}\rangle \ = \ \langle\text{exp}\rangle$$

The recursive applicative program schemes form a set S.

□

Reduction for program schemes proceeds on the expression at the end of a scheme. The new constructs *let*, *if*, and ⟨name⟩ have new reduction rules:

- *let* is reduced according to

$$let\ x \ = \ e'\ in\ e \equiv (\lambda x.e)\ e'$$

- The above definition for program schemes relies on constants *true* and *false* for booleans. The *if* is reduced in the intuitive way:

$$if\ true\ then\ e_1\ else\ e_2 \rightarrow e_1$$
$$if\ false\ then\ e_1\ else\ e_2 \rightarrow e_2$$

- A function name in a lambda term must correspond to an equation; it reduces to the right-hand side of that equation.
- For constants, the "user" must define additional reduction rules, so-called δ reductions which depend on their concrete meaning. For instance, if a scheme were to support integers and addition, there would be constants $\underline{n}$ for all integers n as well as a constant $+$ with rules:

$$+\ \underline{m}\ \underline{n} \rightarrow_\delta \underline{m+n}$$

3.5 Denotational semantics of the lambda calculus

Reduction rules are a convenient way of giving meaning to lambda terms. Unfortunately, they are not suitable as a basis for an implementation of the lambda calculus. Programming language implementation by reduction is inefficient. However, denotational semantics provide the necessary formalism to give precise meaning to a lambda calculus term (or, rather, a recursive program scheme) while still being suitable for the straightforward derivation of an implementation.

3.5.1 Domains for semantics

The central idea in denotational semantics is to assign a mathematical *object* to a program—rather than a behavior as specified by reduction rules. Formulating a denotational semantics for a programming language as a first step requires specifying the computational domains of the semantics. Much theoretical background exists on such domains, but an expository treatment suffices for our purposes.

The first step in forming such domains is to recognize that describing recursion requires computing fixpoints. Computation of fixpoints requires some sort of fixpoint iteration over a mathematical structure with some partial order. The partial order intuitively describes the amount of information in a value; fixpoint iteration

must increase this amount upon every iteration to guarantee that it will compute a meaningful value.

For a domain D, the order will usually be written as $\sqsubseteq_D$. As a starting point for fixpoint iteration (and as a prerequisite for the existence of fixpoints) a semantical domain D must contain a least element element $\bot_D$ (pronounced "bottom") that stands for a non-terminating computation—an undefined value. Domain theory guarantees that recursive equations occurring in semantics actually have a smallest solution.

Of course, realistic computation requires the assembly of larger domains from smaller ones. Here are a few constructions used in the semantics for program schemes:

Primitive domains

- $\mathbb{Z}_\bot$ is the flat domain of integers consisting of $\bot_{\mathbb{Z}_\bot}, 0, 1, -1, \ldots$.
- $\mathbb{T}_\bot$ is the flat domain of booleans consisting of $\bot_\mathbb{T}$, *true*, and *false*. The expression $e \to e_1, e_2$ denotes a conditional: If e denotes *true*, the expression's denotation is that of e_1, in the case of *false*, it is that of e_2, and $\bot$ otherwise.

Lifted domains

- $D_\bot$ denotes a domain D' obtained by adding a new least element $\bot_{D'}$ under a "copy" of D.
- $\mathrm{up}_D : D \to D_\bot$ maps each element of D to the corresponding element of $D_\bot$.
- $\mathrm{down}_D : D_\bot \to D$ maps each element of $D_\bot$ back to the corresponding element of D.

Product domains

- $D_1 \times \cdots \times D_n$ is the Cartesian product of n-tuples. The order of the product is the pointwise extension of the factors. The element of such a product have the form $(e_1, \ldots, e_n)$.
- $D_1 \otimes \cdots \otimes D_n$ denotes the *smash product* obtained from the Cartesian product by identifying all tuples which contain a $\bot$ factor with a new element $\bot$.
- $\mathrm{smash}_D : D_1 \times \cdots \times D_n \to D_1 \otimes \cdots \otimes D_n$ maps each element of a Cartesian product into the corresponding smash product.
- $\mathrm{unsmash}_D : D_1 \otimes \cdots \otimes D_n \to D_1 \times \cdots \times D_n$ maps each element of a smash product into the corresponding Cartesian product.

Sum Domains

- $D_1 + \cdots + D_n$ is the separated sum of the domains D_i whose elements are distinguished copies of the elements of D_i together with new $\bot_{D_1+\cdots+D_n}$.
- $D_1 \oplus \cdots \oplus D_n$ is the coalesced sum domains obtained from the separated sum by identifying all copies of $\bot_{D_i}$ elements with a new element $\bot$.
- in_i^D is an injection from a summand D_i of $D = D_1 \oplus \cdots \oplus D_n$ into D.
- $[e_1, \ldots, e_n] : D_1 \oplus \cdots \oplus D_n \to D'$ is the case analysis of functions $f_i : D_i \to D'$ denoted by the e_i mapping $\mathrm{in}_i^D(x)$ (with $D = D_1 \oplus \cdots \oplus D_n$) to $f_i(x)$.

Function Domains

- $D_1 \to D_2$ is the domain of all continuous functions from D_1 to D_2: $f \sqsubseteq_{D_1 \to D_2} g$ iff $f(x) \sqsubseteq_{D_2} g(x)$ for all $x \in D_1$.
- $\lambda x \in D.e$ denotes a continuous function f given by defining $f(x) := e$ where x ranges over D. (The notational clash with the lambda calculus is unfortunate, but pervasive in the literature.)
- $f\ e$ denotes the result of applying f to e.
- lfp_D is the least fixpoint operator for a domain D which maps a function f in $D \to D$ to the least solution of $x = f(x)$.
- $D_1 \circ\!\!\to D_2$ is the restriction of $D_1 \to D_2$ to strict functions, that is, functions f with $f\ \bot_{D_1} = \bot_{D_2}$.
- $\mathrm{strict}_D : (D_1 \to D_2) \to (D_1 \circ\!\!\to D_2)$ is the functions that maps each continuous function to its strict counterpart.

3.5.2 Semantics of program schemes

Expressions such as $+\ \mathit{false}\ x$ are erroneous. They need a separate domain:

$$\mathit{Error} := \{\mathit{wrong}\}_\bot$$

These domains, together with a domain for functions, form the domain *Val* relevant for the meanings of lambda terms:

$$\begin{aligned} \mathit{Val} &= \mathbb{Z}_\bot \oplus \mathbb{T} \oplus \mathit{Fun} \oplus \mathit{Error} \\ \mathit{Fun} &= \mathit{Val} \to \mathit{Val} \end{aligned}$$

The case analysis construct function induces subdomain-specific variants $\mathrm{case}_S : \mathit{Val} \times (S \to \mathit{Val}) \to \mathit{Val}$. For $\mathbb{Z}_\bot$, it goes like this:

$$\mathrm{case}_{\mathbb{Z}_\bot}(x, f) := [f, \lambda b \in \mathbb{T}.\mathit{wrong}, \lambda g \in \mathit{Fun}.\mathit{wrong}, \lambda e \in \mathit{Error}.\mathit{wrong}](x).$$

For denotational semantics, an additional domain is required to hold *environments* which map variables to values:

$$\mathit{Env} := V \to \mathit{Val}$$

Furthermore, the semantics also deals with environments that map function names to values:

$$\mathit{FEnv} := F \to \mathit{Val}$$

For a denotational semantics for program schemes, assume a function $\alpha \in C \to \mathit{Val}$ which defines meanings for the constants. A fragment of it for $\mathbb{Z}_\bot$ and $\mathbb{T}$ may look as follows:

$$\begin{aligned} \alpha[\![n]\!] &:= \mathrm{in}_{\mathbb{Z}_\bot}\, n \quad n \in \mathbb{Z} \\ \alpha[\![t]\!] &:= \mathrm{in}_{\mathbb{T}}\, t \quad n \in \mathbb{T} \\ \alpha[\![+]\!] &:= \mathrm{in}_{\mathit{Fun}}(\lambda x \in \mathit{Val}.\, \mathrm{in}_{\mathit{Fun}}(\lambda y \in \mathit{Val}.\, \mathrm{case}_{\mathbb{Z}_\bot}(x, \lambda x'.\, \mathrm{case}_{\mathbb{Z}_\bot}(y, \lambda y'.\, \mathrm{in}_{\mathbb{Z}_\bot}(x' + y'))) \end{aligned}$$

The semantics function for terms $[\![_]\!] : E' \to FEnv \to Env \to Val$ is defined by the following structural induction:

$$\begin{aligned}
[\![v]\!]\varphi\rho &:= \rho[\![v]\!] \quad v \in V \\
[\![c]\!]\varphi\rho &:= \alpha[\![c]\!] \quad c \in C \\
[\![f]\!]\varphi\rho &:= \varphi[\![f]\!] \quad f \in F \\
[\![\lambda x.e]\!]\varphi\rho &:= \mathrm{in}_{Fun}(\lambda y \in Val.[\![e]\!]\varphi\rho[x \mapsto y]) \\
[\![(e_1\ e_2)]\!]\varphi\rho &:= \mathrm{case}_{Fun}([\![e_1]\!]\varphi\rho, \lambda g \in Fun.g([\![e_2]\!]\varphi\rho)) \\
[\![let\ v = e'\ in\ e]\!]\varphi\rho &:= [\![e]\!]\varphi\rho[v \mapsto [\![e']\!]\varphi\rho] \\
[\![if\ e\ then\ e_1\ e_2]\!]\varphi\rho &:= \mathrm{case}_{\mathbb{T}}([\![e]\!]\varphi\rho, \lambda b \in \mathbb{T}.b \to [\![e_1]\!]\varphi\rho, [\![e_2]\!]\varphi\rho)
\end{aligned}$$

The above semantics builds on an already-existing meaning function for the defined functions of a program scheme. Of course, this meaning function in turn results from an *application* of the above semantics. The main task to be done for the "lifting" of the expression semantics to program schemes is to describe the element of recursion: a fixpoint application does the trick. If S is the reference program scheme, $S(f)$ for $f \in F$ is to be the right-hand side e of the equation $f = e$ in the scheme. Here is how to compute a function environment for a program scheme S:

$$\Phi(S) := \mathrm{lfp}\ \lambda\varphi \in FEnv.\lambda f \in F.[\![S(f)]\!]\varphi\rho_{empty}$$

Now, it is straightforward to express the semantics of a whole program scheme. Let e_S be the body expression of the program scheme S, and ρ_{empty} an empty environment that maps each name to $\perp$.

$$[\![S]\!] := [\![e_S]\!](\Phi(S))\rho_{empty}$$

The semantics is easily converted into a strict variant by modifying the rules for function creation and *let*:

$$\begin{aligned}
[\![\lambda x.e]\!]\varphi\rho &:= \mathrm{in}_{Fun}(\mathrm{strict}(\lambda y \in Val.[\![e]\!]\varphi\rho[x \mapsto y])) \\
[\![let\ v = e'\ in\ e]\!]\varphi\rho &:= ([\![e']\!]\varphi\rho = \perp) \to \perp, [\![e]\!]\varphi\rho[v \mapsto [\![e']\!]\varphi\rho]
\end{aligned}$$

3.6 Lambda lifting

Notationally, program schemes are already very close to actual functional programs. Yet, their structure is sufficiently simple to make a formal description of their semantics simple and natural: Since explicit recursion can occur only at top level, the fixpoint computation factors out nicely to the outer level.

However, languages like Caml also allow for local recursive definition via the `let rec` construct. This construct also appears in Mini-Caml and its abstract syntax. Re-introducing it into program schemes would introduce numerous complications: The semantics would need to handle recursion at two different places, and the beauty of top-level recursion would be lost. Furthermore, the internal `let rec` is also tedious to implement correctly and efficiently, as it will turn out. Hence, proper translation of Mini-Caml into program schemes will require prior removal of `let rec` via a technique called *lambda lifting*

For the purposes of the discussion in this section, we will temporarily assume an additional *letrec* construct in the syntax:

$\langle$exp$\rangle$ $\longrightarrow$ *letrec* $\langle$binding$\rangle$ $\langle$additional-binding$\rangle$* *in* $\langle$exp$\rangle$
$\langle$binding$\rangle$ $\longrightarrow$ $\langle$var$\rangle$ = $\langle$exp$\rangle$
$\langle$additional-binding$\rangle$ $\longrightarrow$ *and* $\langle$binding$\rangle$

The right-hand side of a *letrec* must be an abstraction. This restriction is mainly technical, but has also found its way into the syntax of Caml.

The new *letrec* extends the reduction semantics for program schemes to also include *letrec*-bound identifiers in the scope of a redex. The idea of lambda lifting is to transform a program scheme with *letrec* into an equivalent one without one. Thus, it is a simple form of a *program transformation*, an important tool in compiler construction.

A good way to understand lambda lifting is to look at a few examples first and develop the general strategy from the insights gained on the way.

3.6.1 Strategies for removing *letrec*

Consider the following term involving *letrec* (ignoring for a moment that it does not make much sense operationally):

$$\mathit{let}\ i\ =\ 5\ \mathit{in}\ \mathit{letrec}\ f\ =\ \lambda x.f\ (+\ i\ i)\ \mathit{in}\ f\ (*\ i\ i)$$

Since this term involves "interior" recursion, and vanilla program schemes only provide top-level recursion, the basic approach to remove the *letrec* must be to "lift" f to the top level:

$$\begin{array}{l} f =\ \lambda x.f\ (+\ i\ i) \\ \mathit{let}\ i\ =\ 5\ \mathit{in}\ f\ (*\ i\ i) \end{array}$$

Unfortunately, the resulting program does not work because $\lambda x.f\ (+\ i\ i)$ has a free variable i, and the lifting has removed the term from the scope of i. Therefore, it is necessary to pass i as an additional parameter to f:

$$\begin{array}{l} f =\ \lambda i.\lambda x.f\ (+\ i\ i) \\ \mathit{let}\ i\ =\ 5\ \mathit{in}\ f\ i\ (*\ i\ i) \end{array}$$

As an afterthought, it may have been more systematic to add the additional parameters first, and then lift them to the top level, thus avoiding an incorrect program at an intermediate stage:

$$\begin{array}{l} \mathit{let}\ i\ =\ 5\ \mathit{in}\ \mathit{letrec}\ f\ =\ \lambda x.f\ (+\ i\ i)\ \mathit{in}\ f\ (*\ i\ i) \\ \Longrightarrow \\ \mathit{let}\ i\ =\ 5\ \mathit{in}\ \mathit{letrec}\ f\ =\ \lambda i.\lambda x.f\ (+\ i\ i)\ \mathit{in}\ f\ i\ (*\ i\ i) \\ \Longrightarrow \\ f\ =\ \lambda i.\lambda x.f\ (+\ i\ i) \\ \mathit{let}\ i\ =\ 5\ \mathit{in}\ f\ i\ (*\ i\ i) \end{array}$$

It has now become visible that this introduction of additional parameters is really just the reverse of η reduction—so-called η *expansion*, and hence clearly a step which preserves the meaning of the program scheme. Also, pulling interior *letrec*-bound expressions to the top level is clearly compatible with β reduction and therefore semantics-preserving.

Unfortunately, it is not always quite sufficient to abstract over all free variables of a *letrec*-bound term. Consider the following, somewhat more involved example:

$$\begin{array}{l} \mathit{let}\ a\ =\ \ldots \mathit{in} \\ \mathit{let}\ b\ =\ \ldots \mathit{in} \\ \mathit{letrec}\ f\ =\ \lambda x.\ldots a \ldots g \ldots \\ \mathit{and}\ g\ =\ \lambda y.\ldots b \ldots f \ldots \\ \mathit{in}\ \ldots f \ldots g \ldots \end{array}$$

Here, a is free in f, and b is free in g. A naive application of the η expansion strategy produces the following result:

$$\begin{aligned}
&\textit{let } a = \ldots \textit{in} \\
&\textit{let } b = \ldots \textit{in} \\
&\textit{letrec } f = \lambda a.\lambda x.\ldots a \ldots g\ b \ldots \\
&\textit{and } g = \lambda b.\lambda y.\ldots b \ldots f\ a \ldots \\
&\textit{in } \ldots f\ a \ldots g\ b \ldots
\end{aligned}$$

The result still has the same meaning as the original, but lifting is not possible yet: The η expansion has introduced new free variables in the bodies of f and g: b is now free in f, and a is free in g. Consequently, another application of the η expansion step is necessary:

$$\begin{aligned}
&\textit{let } a = \ldots \textit{in} \\
&\textit{let } b = \ldots \textit{in} \\
&\textit{letrec } f = \lambda b.\lambda a.\lambda x.\ldots a \ldots g\ a\ b \ldots \\
&\textit{and } g = \lambda a.\lambda b.\lambda y.\ldots b \ldots f\ b\ a \ldots \\
&\textit{in } \ldots f\ b\ a \ldots g\ a\ b \ldots
\end{aligned}$$

Now, finally, the bodies both contain no more free variables, and lifting is possible:

$$\begin{aligned}
&f = \lambda b.\lambda a.\lambda x.\ldots a \ldots g\ a\ b \ldots \\
&g = \lambda a.\lambda b.\lambda y.\ldots b \ldots f\ b\ a \ldots \\
&\textit{let } a = \ldots \textit{in} \\
&\textit{let } b = \ldots \textit{in} \\
&\ldots f\ b\ a \ldots g\ a\ b \ldots
\end{aligned}$$

Unfortunately, there is no limit on how often the η expansion needs to be repeated until a fixpoint is reached. Obviously, performing this step successively repeatedly in a compiler is a costly matter just to cater to such a non-consequential language construct. Fortunately, a more systematic approach to the problem yields a more direct algorithm which is efficient enough in practice.

3.6.2 An algorithm for lambda lifting

The main goal for a lambda lifting algorithm is that is that it should transform the program only once. Hence, the algorithm needs to compute the set of variables over which abstraction is necessary. A straightforward approach to this is to formulate the constraints on these free variable sets as set equations and then solve those.

A prerequisite is that all identifiers in the program scheme must be unique: The program may bind no identifier twice. This simplifies the formulation of the algorithm, and generally prevents a few implementation headaches that have to do with inadvertent name capture problems.

The subject of the algorithm is a subterm e of a program scheme which has the following form:

$$\begin{aligned}
&\textit{letrec} \quad f_1 = e_1 \\
&\ldots \\
&\textit{and} \quad f_n = e_n \\
&\textit{in} \quad e
\end{aligned}$$

To prepare the f_i for lifting, abstraction is necessary over a set of variables A_{f_i}, and the following must hold:

$$A_{f_i} = \text{free}(e_i) \cup \bigcup \{A_f \mid f \in \text{free}(e_i), f \textit{ letrec}\text{-bound}\}$$

Using this equation and solving the resulting equation system over a whole program is still somewhat costly. However, it is possible to subdivide A_{f_i} and develop that into more manageable equations. There are

- the abstracted variables of all *letrec*-bound functions referenced from f_i bound in an outer scope, and
- the abstracted variables of all functions f_j bound in the same *letrec* as f_i itself.

The set of functions f_j referenced from f_i also receives a name:

$$m_{f_i} := \{f_j \in \text{free}(e_i)\}$$

Also, there are the functions referenced from f_i bound in some outer scope:

$$r_{f_i} := \{f \in \text{free}(e_i) \mid f \text{ } \textit{letrec}\text{-bound}\} \setminus m_{f_i}$$

Now, the revised equation for A_{f_i} goes as follows:

$$A_{f_i} = \text{free}(e_i) \cup \bigcup\{A_f \mid f \in r_{f_i}\} \cup \bigcup\{A_f \mid f \in m_{f_i}\}$$

In the revised equation, the recursion is limited to terms coming from right-hand sides of the same *letrec*. Since internal *letrec*s are usually not overly big, the resulting equation system is usually quite manageable. Also, the resulting equation system represents (for each *letrec* separately) a so-called *pure inclusion problem* for which efficient algorithms exist [DP82, WM92].

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Boc76] G. V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19:55–62, 1976.

[Cha87] Nigel P. Chapman. *LR parsing: theory and practice.* Cambridge University Press, Cambridge, UK, 1987.

[DeR69] Franklin L. DeRemer. *Practical Translators for LR(k) Parsers.* PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Ma., 1969.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[DP82] Franklin DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

[DS95] Charles Donnelly and Richard Stallman. *Bison—The YACC-compatible Parser Generator.* Free Software Foundation, Boston, MA, November 1995. Part of the Bison distribution.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995.

[FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *Letters on Programming Languages and Systems*, 1(3):213–226, 1992.

[Fis93] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993.

[Ive86] Fred Ives. Unifying view of recent LALR(1) lookahead set algorithms. *SIGPLAN Notices*, 21(7):131–135, July 1986. Proceedings of the SIGPLAN'86 Symposium on Compiler Construction.

[Ive87a] Fred Ives. An LALR(1) lookahead set algorithm. Unpublished manuscript, 1987.

[Ive87b] Fred Ives. Response to remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(8):99–104, August 1987.

[Joh75] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Lee93] Rene Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, Boston, 1993.

[PC87] Joseph C.H. Park and Kwang-Moo Choe. Remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(4):30–32, April 1987.

[PCC85] Joseph C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, January 1985.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Spe94] Michael Sperber. Attribute-directed functional LR parsing. Unpublished manuscript, October 1994.

[SSS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II (LR(k) and LL(k) Parsing) of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1990.

[ST95] Michael Sperber and Peter Thiemann. The essence of LR parsing. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, June 1995. ACM Press.

[ST00] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung*. Lehrbuch. Springer-Verlag, Berlin, Heidelberg, 1992.