$$\varphi_{\mathtt{p}}(\mathtt{x},\mathtt{y}) = \varphi_{\varphi_{\mathtt{mix}}(\mathtt{p},\mathtt{x})}(\mathtt{y})$$

which is just the ‘mix equation’ in another guise.

The standard proof of the s-m-n property for Turing machines in essence uses a trivial construction that *never* gains efficiency, but this suffices for the purposes of recursive function theory. Efficiency is very important in applications though, so partial evaluation may be regarded as the quest for efficient implementations of the s-m-n theorem.

Clearly each of the languages we have studied in earlier chapters is an acceptable programming system.

The traditional usage of natural numbers in recursive function theory is simple, abstract and elegant, but involves a high computational price: all program structures and non-numeric data must be encoded by means of Gödel numbers, and operations must be done on encoded values. Letting *programs and data* have the same form allows the theory to be developed without the trick of Gödel numbering. This is a substantial advantage when doing recursion theoretic constructions such as needed to prove Kleene’s s-m-n and Second Recursion theorems, and leads to faster constructed programs.

16.2 Types for interpreters, compilers, and partial evaluators

High-level operations in programming languages

Programming languages of higher and higher abstraction levels have evolved since the first years of computing, when programming languages were just symbolic codes reflecting the computer’s architecture. Due to higher level basic operations, modern functional languages allow a mathematical style of thinking while programming, for example using function composition, partial function application, set comprehension, and pattern matching. This is possible since these operations are all in the so-called ‘constructable’ part of mathematics, known to give computable results when applied to computable arguments.

Many operations on mathematical objects can be faithfully realized by corresponding operations on symbolic expressions. Classically, algebraic manipulation is used to organize arithmetic computations more efficiently — possible because algebra abstractly but correctly describes concrete operations on numbers. On digital computers, symbolic operations are specified by textual objects, i.e. programs and their subexpressions. The term ‘symbolic computation’ often refers to algebraic manipulations when realized on the computer, but can be interpreted more broadly to describe the entire theme of this book[2].

[2]Much of this material is from [132].

Operations on functions and programs
Two useful operations on (mathematical) functions are:

- *Composition* of f with g as in Chapter 2, written as $f; g$ or $g \circ f$.
- *Function specialization* of $f(x, y)$, obtaining for example a one-argument function $f_{|x=a}(y) = f(a, y)$ by 'freezing' x to a fixed value a.

Corresponding symbolic operations on programs (assuming for concreteness that they are written in the λ-calculus):

Symbolic composition. The *symbolic composition* of expressions e_f and g could be expression $\lambda x.e_g(e_f(x))$.

Partial evaluation. The specialization of function f to $x = a$ can be realized symbolically as the program $\lambda y.e_f(a, y)$. In the context of recursive function theory this is Kleene's s-m-n theorem [149,226], and its efficient realization is of course the theme of this book.

Efficient operations on programs
The symbolic operations above, while computable, do not lead to particularly efficient programs. For example, the program above realizing function composition, $\lambda x.e_g(e_f(x))$, is no faster than just running the two programs from which it is constructed, one after the other. A main theme of this book is the *efficient* implementation of program operations that realize mathematical operations.

Deforestation as in Chapter 17 symbolically realizes function composition, and partial evaluation is of course a symbolic realization of function specialization.

Data and program types
How can one describe the types of operations on symbolic expressions? A symbolic composition operator (for example) takes programs $\mathtt{p}_f$, $\mathtt{p}_g$ computing $f : A \to B$ and $g : B \to C$ (respectively) into a program `q` computing $f; g : A \to C$. The same symbolic composer works, *independently* of A, B, C. Thus a symbolic operation should in some sense be polymorphic [185] in the types of its arguments.

A more subtle problem is the 'level shift' that occurs when going from a program text `p` to the function $[\![\mathtt{p}]\!]_{\mathtt{X}}$ it denotes when regarded as a program in language `X`. To describe this symbolically we assume assume given a fixed collection of programming languages generically called `X`, and extend the usual concept of type according to the following syntax:

$$t : \textit{type} ::= \underline{t}_{\mathtt{X}} \mid \textit{firstorder} \mid t \times t \mid t \to t$$

Type *firstorder* describes values in `D`, for example S-expressions, and function types and products are as usual. For each language `X` and type t we have a type $\underline{t}_{\mathtt{X}}$,

$$\frac{exp : \underline{t}_{\mathtt{X}}}{[\![exp]\!]_{\mathtt{X}} : t} \qquad \frac{exp_1 : t_2 \to t_1, \quad exp_2 : t_2}{exp_1(exp_2) : t_1}$$

$$\frac{}{\mathit{firstordervalue} : \mathit{firstorder}} \qquad \frac{exp : \underline{t}_{\mathtt{X}}}{exp : \mathit{firstorder}}$$

Figure 16.1: Some type inference rules for closed expressions.

meaning the type of all `X`-programs which denote values of type t. For example, atom `1066` has type *firstorder*, and Scheme program `(quote 1066)` has type $\underline{firstorder}_{\mathtt{Scheme}}$.

The subscript `X` will often be dropped when the language being discussed is the standard implementation language, always called `L`.

Semantics of types. The meaning of type expression t is a set $\mathcal{T}(t)$ defined as follows, where $[A \to B]$ is the set of all functions from A to B:

$$\begin{array}{lcl}
\mathcal{T}(\mathit{firstorder}) & = & \mathtt{D} \\
\mathcal{T}(t_1 \to t_2) & = & [\mathcal{T}(t_1) \to \mathcal{T}(t_2)] \\
\mathcal{T}(t_1 \times t_2) & = & \{(v_1, v_2) \mid v_1 \in \mathcal{T}(t_1), v_2 \in \mathcal{T}(t_2)\} \\
\mathcal{T}(\underline{t}_{\mathtt{X}}) & = & \{\mathtt{p} \in \mathtt{D} \mid [\![\mathtt{p}]\!]_{\mathtt{X}} \in \mathcal{T}(t)\}
\end{array}$$

Polymorphism. We shall also allow *polymorphic* type expressions to be written containing *type variables* $\alpha, \beta, \gamma, \ldots$. Such a polymorphic type will always be understood as standing for the set of all the *monomorphic instances* obtained from it by consistently replacing type variables by variable-free type expressions. To emphasize this, we will often (informally) quantify type variables universally, e.g. $\forall\alpha.(\alpha \to \alpha)$. The result of replacing type variables is called an *instance* of the polymorphic type.

Type inference rules. Figure 16.1 contains some rules sufficient to infer the types involved in program runs, i.e. evaluations of closed expressions. Note that an object `p` of type $\underline{t}_{\mathtt{X}}$ is a program text and thus *in itself* a value in `D`, i.e. $\underline{t}_{\mathtt{X}}$ denotes a subset of `D`. On the other hand, `p`'s meaning $[\![\mathtt{p}]\!]_{\mathtt{X}}$ may be any value, for example a higher-order function.

Program equivalence. It is important to be able to say when two programs `p, q` $\in D$ are computationally equivalent. In recent years two views have developed, *semantic equivalence* and *observational equivalence* [244,219][3]. Both concepts make sense in our framework, defined as follows. Let `p, q` $\in D$. Then `p` and `q` are

[3]A denotational semantics is said to be *fully abstract* with respect to an operational semantics if observational and semantic equivalence are the same.

semantically equivalent if $[\![\mathtt{p}]\!] = [\![\mathtt{q}]\!]$

observationally equivalent if $[\![\mathtt{p}]\!] \approx [\![\mathtt{q}]\!]$, where we define $f \approx g$ to mean that for all $n \geq 0$ and for all $\mathtt{d}_1, \ldots, \mathtt{d}_n \in D$ and $\mathtt{d} \in D$,

$$(f\mathtt{d}_1 \ldots \mathtt{d}_n = \mathtt{d}) \text{ if and only if } (g\mathtt{d}_1 \ldots \mathtt{d}_n = \mathtt{d})$$

The first definition is the easier to formulate, but a case can be made that the second is more relevant in practice. The reason is that establishing the first requires verifying equality between two elements of a semantic function domain. This can be a tricky task, and computationally speaking too strict if the semantic function $[\![\_]\!]$ is not fully abstract.

The second definition is a version of the observational equivalence studied by Plotkin, Milner, and others, limited to first-order applicative contexts. It only involves assertions that can in principle be verified by running the program on first-order inputs and observing its first-order outputs or nontermination behaviour.

16.2.1 Efficient symbolic composition

Symbolic composition can be described as commutativity of the diagram in Figure 16.2, where α, β, γ range over all types. We now list some examples of symbolic composition, and discuss what is saved computationally.

Vector spaces and matrix multiplication. Suppose M, N are $n \times n$ matrices over (for example) the real numbers $\mathcal{R}$. Each determines a linear transformation, e.g. $[\![M]\!] : \mathcal{R}^n \to \mathcal{R}^n$. If $M \cdot N$ is their matrix product, then

$$[\![M \cdot N]\!](\vec{w}) = [\![M]\!]([\![N]\!](\vec{w}))$$

The composite linear transformation can be computed in either of two ways:

- by applying first N and then M to $\vec{w}$, taking time $2n^2$; or
- by first multiplying M and N (time n^3 by the usual algorithm), and applying the result to $\vec{w}$ (time n^2)

It may be asked: what if anything has been saved? The answer is: nothing, if the goal is only to transform a single vector, since the second time always exceeds the first. There is, however, a net saving if more than n vectors are to be transformed since the matrix product need only be computed once.

The moral: an operation so familiar as matrix multiplication can be thought of as symbolic composition, and composition can save computational time.

Other examples of efficient symbolic composition include the fact that two *finite state transducers* can be combined into one with no intermediate symbols; *deforestation*, seen in Chapter 17; and composition of *derivors* or of *attribute coupled grammars*. The latter two can be used automatically to combine multipass compiler phases into a single phase.

$$\begin{tikzcd}
\underline{(\alpha \to \beta)} \times \underline{(\beta \to \gamma)} \arrow[r, "\underline{;}"] \arrow[d, "[\![-]\!] \times [\![-]\!]"'] & \underline{(\alpha \to \gamma)} \arrow[d, "[\![-]\!]"] & \text{Syntactic composition} \\
(\alpha \to \beta) \times (\beta \to \gamma) \arrow[r, ";"] & (\alpha \to \gamma) & \text{Semantic composition}
\end{tikzcd}$$

Figure 16.2: Symbolic composition.

16.2.2 Symbolic function specialization = partial evaluation

Specializing (also called *restricting*) a two-argument function $f(x, y) : \alpha \times \beta \to \gamma$ to $x = a$ gives the function $f_{|x=a}(y) = f(a, y)$. Function specialization thus has polymorphic type

$$\texttt{fs} : (\alpha \times \beta \to \gamma) \times \alpha \to (\beta \to \gamma)$$

Partial evaluation is the symbolic operation corresponding to function specialization. Using `peval` = $[\![$`mix`$]\!]$ to denote the partial evaluation function, its correctness is expressed by commutativity of the diagram in Figure 16.3. Partial evaluation has polymorphic type

$$\texttt{peval} : \underline{(\alpha \times \beta \to \gamma)} \times \alpha \to \underline{(\beta \to \gamma)}$$

Redefinition of partial evaluation
The description of `peval` can be both simplified and generalized by writing the functions involved in *curried* form[4]. This gives `peval` a new polymorphic type:

$$\texttt{peval} : \underline{\alpha \to (\beta \to \gamma)} \to \alpha \to \underline{\beta \to \gamma}$$

which is an instance of a more general polymorphic type:

$$\texttt{peval} : \underline{\rho \to \sigma} \to \rho \to \underline{\sigma}$$

Maintaining our emphasis on observable values, we will require ρ to be a first-order type (i.e. *base* or a type $\underline{t}_{\mathrm{L}}$).

[4]The well-known 'curry' isomorphism on functions is $(\alpha \to (\beta \to \gamma)) \simeq (\alpha \times \beta \to \gamma)$.

$$\begin{tikzcd}
\underline{(\alpha \times \beta \to \gamma)} \times \alpha \arrow[r, "\texttt{peval}"] \arrow[d, "\llbracket\texttt{-}\rrbracket \times \text{Identity}"] & \underline{(\beta \to \gamma)} \arrow[d, "\llbracket\texttt{-}\rrbracket"] \\
(\alpha \times \beta \to \gamma) \times \alpha \arrow[r, "\texttt{fs}"] & (\beta \to \gamma)
\end{tikzcd}$$

Figure 16.3: Function specialization and partial evaluation.

Remark. The second input to `peval` is a *value* of type ρ, and not *representation* of a value. In practice, especially if `peval` is programmed in a strongly typed language, one may need to work with representations rather than directly with values. We ignore this aspect partly because of notational complexity, and partly because the use of representations leads quickly to problems of size explosion when dealing with representations of representations of .... This problem has been addressed by Launchbury [169], and appears in Chapter 11.

The mix equation revisited. `mix` $\in D$ is a *partial evaluator* if for all `p, a` $\in D$,

$$\llbracket \texttt{p} \rrbracket\ \texttt{a} \approx \llbracket \llbracket \texttt{mix} \rrbracket\ \texttt{p}\,\texttt{a} \rrbracket$$

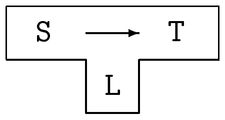
Thus for any $n+1$ first-order values $\texttt{d}$, $\texttt{d}_1,\ldots,\texttt{d}_n$, we have $\llbracket \texttt{p} \rrbracket\ \texttt{a}\,\texttt{d}_1\ldots\texttt{d}_n = \texttt{d}$ if and only if $\llbracket \llbracket \texttt{mix} \rrbracket\ \texttt{p}\,\texttt{a} \rrbracket\,\texttt{d}_1\ldots\texttt{d}_n = \texttt{d}$.

16.2.3 Compiler and interpreter types

Similarly, the definitions of interpreter and compiler of Section 3.1.1 may be elegantly restated, a little more generally than before:

$$\boxed{\begin{array}{c}\texttt{S}\\ \texttt{L}\end{array}} = \{\texttt{int} \mid \llbracket \texttt{s} \rrbracket_{\texttt{S}} \approx \llbracket \texttt{int} \rrbracket_{\texttt{L}} \texttt{s}\}$$

and



$$= \{\ \texttt{comp}\ \mid\ [\![\texttt{s}]\!]_{\texttt{S}} \approx [\![[\![\texttt{comp}]\!]_{\texttt{L}}\texttt{s}]\!]_{\texttt{T}}\}$$

Can an interpreter be typed?
Suppose we have an interpreter up for language L, and written in the same language — a universal program or self-interpreter. By definition up must satisfy $[\![\texttt{p}]\!] \approx [\![\texttt{up}]\!]$ p for any L-program p. Consequently as p ranges over all L-programs, $[\![\texttt{up}]\!]$ p can take on *any* program-expressible type. A difficult question arises: is it possible to define the type of up non-trivially[5]?

A traditional response to this problem has been to write an interpreter in an *untyped* language, e.g. Scheme. This has the disadvantage that it is hard to verify that the interpreter correctly implements the type system of its input language (if any). The reason is that there are two classes of possible errors: those caused by errors in the program being interpreted, and those caused by a badly written interpreter. Without a type system it is difficult to distinguish the one class of interpret-time errors from the other.

Well-typed language processors
Given a source S-program denoting a value of some type t, an S-interpreter should return a value whose type is t. From the same source program, a compiler should yield a target language program whose T-denotation is identical to its source program's S-denotation. This agrees with daily experience—a compiler is a meaning-preserving program transformation, insensitive to the type of its input program (provided only that it *is* well-typed). Analogous requirements apply to partial evaluators.

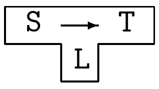
A well-typed interpreter is required to have many types: one for every possible input program type. Thus to satisfy these definitions we must dispense with type *unicity*, and allow the type of the interpreting program not to be uniquely determined by its syntax.

Compilers must satisfy an analogous demand. One example: $\lambda\texttt{x.x}$ has type $\underline{\texttt{t}}_{\texttt{L}} \rightarrow \underline{\texttt{t}}_{\texttt{L}}$ for all types t. It is thus a trivial but well-typed compiling function from L to L. (Henceforth we omit the subscript L.)

A well-typed partial evaluator can be applied to any program p accepting at least one first-order input, together with a value a for p's first input. Suppose p has type $\underline{\rho \rightarrow \sigma}$ where ρ is first-order and $\texttt{a} \in [\![\rho]\!]$. Then $[\![\texttt{mix}]\!]$ p a is a program $\texttt{p}_{\texttt{a}}$ whose result type is σ, the type of $[\![\texttt{p}]\!]\texttt{a}$. Thus $\texttt{p}_{\texttt{a}}$ has type $\underline{\sigma}$.

[5]This question does not arise at all in classical computability theory since there is only one data type, the natural numbers, and all programs denote functions on them. On the other hand, computer scientists are unwilling to code all data as numbers and so demand programs with varying input, output and data types.

1. Interpreter `int` $\in \boxed{\begin{array}{c}\texttt{S}\\ \texttt{L}\end{array}}$ is *well-typed* if it has type[6] $\forall\alpha\,.\,\underline{\underline{\alpha}_{\texttt{S}} \to \alpha}$.

2. Compiler `comp` $\in$  is *well-typed* if it has type $\forall\alpha\,.\,\underline{\underline{\alpha}_{\texttt{S}} \to \underline{\alpha}_{\texttt{T}}}$.

3. A partial evaluator `mix` is *well-typed* if it has type $\forall\rho\,.\,\forall\sigma\,.\,\underline{\underline{\underline{\rho \to \sigma}} \to \rho \to \underline{\sigma}}$, where ρ ranges over first-order types.

Remark. The definition of a well-typed interpreter assumes that all observable `S`-types are also `L`-types. Thus it does not take into account the possibility of encoding `S`-values.

16.2.4 Self-application and types

Definitions involving self-application often (and rightly) cause concern as to their well-typedness. We show here that natural types for `mix`-generated compilers and target programs (and even `cogen` as well) can be deduced from the few type rules of Figure 16.1. Let `source`: $\underline{\alpha}_{\texttt{S}}$ be an `S`-program denoting a value of type α.

First Futamura projection
We wish to find the type of `target` $= \llbracket\texttt{mix}\rrbracket$ `int source`. The following inference concludes that the target program has type $\underline{\alpha} = \underline{\alpha}_{\texttt{L}}$, i.e. that it is an `L` program of the same type as the source program. The inference uses only the rules of Figure 16.1 and instantiation of polymorphic variables.

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{mix} : \underline{\underline{\underline{\rho\to\sigma}}\to\rho\to\underline{\sigma}}}{\llbracket\texttt{mix}\rrbracket : \underline{\underline{\rho\to\sigma}}\to\rho\to\underline{\sigma}}}{\llbracket\texttt{mix}\rrbracket : \underline{\underline{\alpha}_{\texttt{S}}\to\alpha}\to\underline{\alpha}_{\texttt{S}}\to\underline{\alpha}} \quad \texttt{int} : \underline{\underline{\alpha}_{\texttt{S}}\to\alpha}}{\llbracket\texttt{mix}\rrbracket\,\texttt{int} : \underline{\alpha}_{\texttt{S}}\to\underline{\alpha}} \quad \texttt{source} : \underline{\alpha}_{\texttt{S}}}{\llbracket\texttt{mix}\rrbracket\,\texttt{int}\,\texttt{source} : \underline{\alpha}}$$

Second Futamura projection
The previous inference showed that $\llbracket\texttt{mix}\rrbracket$ `int` has the type of a compiling *function*, though it is not a compiler *program*. We now wish to find the type of `compiler` =

[6]As a consequence, $\llbracket\texttt{int}\rrbracket_{\texttt{L}}$ has type $\forall\alpha\,.\,\underline{\alpha}_{\texttt{S}} \to \alpha$, that is to say type $\underline{t}_{\texttt{S}} \to t$ for every type t.

$[\![\texttt{mix}]\!]\,\texttt{mix}\,\texttt{int}$. It turns out to be notationally simpler to begin with a program `p` of more general type $\underline{\alpha\to\beta}$ than that of `int`.

$$\dfrac{\dfrac{\dfrac{\texttt{mix} : \underline{\underline{\underline{\rho\to\sigma}\to\rho\to\underline{\sigma}}}}{[\![\texttt{mix}]\!] : \underline{\rho\to\sigma}\to\rho\to\underline{\sigma}}}{[\![\texttt{mix}]\!] : \underline{\underline{\underline{\underline{\alpha\to\beta}}\to\alpha\to\underline{\underline{\beta}}}}\to\underline{\alpha\to\beta}\to\underline{\alpha\to\underline{\beta}}} \quad \texttt{mix} : \underline{\underline{\alpha\to\beta}\to\alpha\to\underline{\beta}}}{[\![\texttt{mix}]\!]\,\texttt{mix} : \underline{\alpha\to\beta}\to\underline{\alpha\to\underline{\beta}}} \quad \texttt{p} : \underline{\alpha\to\beta}}{[\![\texttt{mix}]\!]\,\texttt{mix}\,\texttt{p} : \underline{\alpha\to\underline{\beta}}}$$

Some interesting substitution instances. Compilers can be generated by the second Futamura projection: $\texttt{compiler} = [\![\texttt{mix}]\!]\,\texttt{mix}\,\texttt{int}$. The type of `int` is $\underline{\underline{\delta}_{\mathtt{S}}\to\delta}$, an instance of the type assigned to `p` above. By the same substitution we have $\texttt{compiler} : \underline{\underline{\delta}_{\mathtt{S}}\to\underline{\delta}}$. Moreover, δ was chosen arbitrarily, so $[\![\texttt{compiler}]\!] : \forall\delta\,.\,\underline{\delta}_{\mathtt{S}}\to\underline{\delta}$ as desired.

The type of a compiler generator. Even `cogen` can be given a type, namely $\underline{\underline{\alpha\to\beta}\to\alpha\to\underline{\beta}}$ by exactly the same technique; but the tree is rather complex. One substitution instance of `cogen`'s type is the conversion of an interpreter's type into that of a compiler.

The type of $[\![\texttt{cogen}]\!]$ is thus $\underline{\alpha\to\beta}\to\underline{\alpha\to\underline{\beta}}$, which looks like the type of the identity function(!) but with some underlining. It is substantially different, however, in that it describes program generation. Specifically

1. $[\![\texttt{cogen}]\!]$ transforms a two-input program `p` into another program `p-gen`, such that for any $\texttt{a}\in D$
2. $\texttt{p}' = [\![\texttt{p-gen}]\!]\texttt{a}$ is a program which
3. for any $\texttt{d}_1,\ldots,\texttt{d}_n \in D$ computes

$$[\![\texttt{p}']\!]\texttt{d}_1\ldots\texttt{d}_n \approx [\![\texttt{p}]\!]\,\texttt{a}\,\texttt{d}_1\ldots\texttt{d}_n$$

One could even describe the function $[\![\texttt{cogen}]\!]$ as an *intensional version of currying*, one that works on program texts instead of on functions. To follow this, the type of $[\![\texttt{cogen}]\!]$ has as a substitution instance

$$[\![\texttt{cogen}]\!] : \underline{\alpha\to(\beta\to\gamma)}\to\underline{\alpha\to\underline{\beta\to\gamma}}$$

In most higher-order languages it requires only a trivial modification of a program text with type $(\alpha \times \beta \to \gamma)$ to obtain a variant with type $(\alpha \to (\beta \to \gamma))$, and with the same (or better) computational complexity. So a variant `cogen′` could be easily constructed that would first carry out this modification on its program input, and then run `cogen` on the result. The function computed by `cogen′` would be of type:

$$[\![\mathtt{cogen'}]\!] : \underline{\alpha \times \beta \to \gamma} \to \underline{\alpha \to \underline{\beta \to \gamma}}$$

which is just the type of the curry transformation, plus some underlining.

16.3 Some research problems

Exercise 16.1 Informally verify type correctness of a simple interpreter with integer and boolean data □

Exercise 16.2 Figure 16.1 contains no *introduction* rules for deducing that any programs at all have types of form $\underline{t}_{\mathtt{L}}$. Problem: for a fixed programming language, find type inference rules appropriate for showing that given programs have given types. □

Exercise 16.3 For a familiar language, e.g. the λ-calculus, formulate a set of type inference rules that is sufficiently general to verify type correctness of a range of compilers, interpreters and partial evaluators. □

Exercise 16.4 Find a suitable model theory for these types (domains, ideals, etc.). The type semantics given earlier uses ordinary sets, but for computational purposes it is desirable that the domains be ω-algebraic, and that the values manipulated by programs should only range over computable elements. □