Journal of Functional Programming 1 (4): 459–494, October 1991

Using transformations in the implementation of higher-order functions

HANNE RIIS NIELSON AND FLEMMING NIELSON

Computer Science Department, Aarhus University, Denmark

Abstract

Traditional functional languages do not have an explicit distinction between binding times. It arises implicitly, however, as one typically instantiates a higher-order function with the arguments that are known, whereas the unknown arguments remain to be taken as parameters. The distinction between '*known*' and '*unknown*' is closely related to the distinction between *binding times* like '*compile-time*' and '*run-time*'. This leads to the use of a combination of *polymorphic type inference* and *binding time analysis* for obtaining the required information about which arguments are known.

Following the current trend in the implementation of functional languages we then transform the run-time level of the program (*not* the compile-time level) into *categorical combinators*. At this stage we have a natural distinction between two kinds of program transformations: *partial evaluation*, which involves the compile-time level of our notation, and *algebraic transformations* (i.e., the application of algebraic laws), which involves the run-time level of our notation.

By reinterpreting the combinators in suitable ways we obtain specifications of *abstract interpretations* (or data flow analyses). In particular, the use of combinators makes it possible to use a general framework for specifying both *forward* and *backward analyses*. The results of these analyses will be used to enable program transformations that are not applicable under all circumstances.

Finally, the combinators can also be reinterpreted to specify *code generation* for various (abstract) machines. To improve the efficiency of the code generated, one may apply abstract interpretations to collect extra information for guiding the code generation. Peephole optimizations may be used to improve the code at the lowest level.

Capsule review

Many different techniques are needed in an optimising compiler – constant folding, program transformation, semantic analysis and optimisation, code generation and so on. This paper proposes a uniform framework for all these activities. Every computation in the program to be compiled is classified as either compile-time or run-time, and the run-time parts are translated into an easily manipulable form – categorical combinators. The Nielsons set up a framework for defining operations on these intermediate forms, and show how it can be used to define concisely two forwards analyses (strictness, constant propagation), a backwards analysis (liveness), and even code generation. Both compile- and run-time parts can also be transformed. Every technique presented is illustrated using a simple running example, and the emphasis throughout is on intuition rather than absolute formality.

Hanne and Flemming Nielson have been engaged for many years in a research program in the area of semantics directed compilation, which has heavily influenced this paper. Much of this work is referred to here, and this paper will also be useful as a clear introduction to and overview of this substantial body of research.

1 Introduction

The *functional programming style* is closely related to the use of higher-order functions. In particular, the functional programming style suggests that many function definitions are instances of the same general computational pattern, and that this pattern is defined by a higher-order function. The various instances of the pattern are then obtained by supplying the higher-order function with some of its arguments.

One of the benefits of this programming style is the reuse of function definitions and, more importantly, the reuse of properties proved to hold for them: usually, a property of a higher-order function carries over to an instance by verifying that the arguments satisfy some simple properties.

One of the disadvantages is that the efficiency in a compiler that reuses code generated for higher-order functions is often rather poor. The reason for this is that when generating code for the higher-order function it is impossible to make any assumptions about its arguments and to optimize the code accordingly. Also, conventional machine architectures make it rather costly to use functions as data.

We shall therefore be interested in *transforming* instances of higher-order functions into functions that can be implemented more efficiently. The key observation in the approach to be presented here is that an instance of a higher-order function is a function where some of the arguments are known and others are not. To be able to exploit this we shall introduce an *explicit* distinction between *known* and *unknown* values or, using traditional compiler terminology, between *compile-time* entities and *run-time* entities. This leads to the following approach to implementing functional languages

- annotate the programs with *type information* so that they can be uniquely typed in a monotyped type system (Section 3);
- annotate the programs with *binding time information* so that there is an explicit distinction between the computations that involve known (compile-time) data and those that involve unknown (run-time) data (Section 4);
- transform the programs into *combinator form* so that the computations involving unknown (run-time) data are expressed as combinators (Section 5);
- specify various abstract interpretations by *reinterpreting* the combinators and use the results to enable *program transformations* that are not applicable under all circumstances (Sections 6, 7 and 8);
- specify a simple-minded code generation scheme for an abstract machine by *reinterpreting* the combinators (Section 9); and
- specify an abstract interpretation by reinterpreting the combinators and use this to improve the code generation (Section 10).

We illustrate our approach on an example program in the enriched λ-calculus, to be presented in Section 2. We assume that the semantics is *non-strict*, i.e., lazy, although most of the analyses and transformations equally well apply to a language with a strict semantics. Finally, Section 11 contains a discussion of our approach including efficiency, correctness and automation.

2 Operations made explicit

In Miranda* (Turner, 1985) the functions `reduce` and `sum` may be written as

```
reduce f u = g
           where g [ ] = u
                 g (x: xs) = f x (g xs)
sum = reduce (+) 0
```

The left-hand side of an equation specifies the name of the function and a list of patterns for its parameters, and the right-hand side is the body of the function. If more than one equation is given for the same function (as is the case for g), there is an *implicit* conditional testing the form of the patterns in the parameter list. Recursion is left *implicit* because a function name on the right-hand side of an equation that defines that function name indicates recursion (as is the case for g). Finally, function application is left *implicit* as the function is just juxtaposed with its argument(s).

A similar equational definition of functions is allowed in Standard ML (Milner, 1984). But here one has the possibility of making some of the implicit operations or concepts more explicit as is illustrated in

```
val reduce = fn f => fn u => let val rec g = fn xs =>
                                  if xs = [ ] then u
                                  else f (hd xs) (g (tl xs))
                             in g end;
val sum = reduce (fn x => fn y => x+y) 0;
```

Function abstraction is now expressed *explicitly* by the construct `fn... =>...` and the recursive structure of g is expressed by the *explicit* occurrence of `rec`. Also, the test on the form of the list argument is expressed *explicitly*.

Our approach requires *all* the operations to be expressed explicitly. We shall therefore define a small language, an *enriched λ-calculus*, that captures a few of the more important constructs present in modern functional languages like Miranda and Standard ML, but in an *explicit* way.

The general form of a program in the enriched λ-calculus is a sequence of definitions (prefixed by `DEF`) followed by an expression (prefixed by `VAL`) and its overall type (prefixed by `HAS`). In the enriched λ-calculus we write the program `sum` as

```
DEF reduce = λf.λu. fix (λg.λxs.  if isnil xs then u
                                   else f (hd xs) (g (tl xs)))
VAL reduce (λx.λy. +(⟨x,y⟩)) (0)
HAS Int list→Int
```

Here we use the special parentheses (and) for the explicit function application.

* Miranda is a trademark of Research Software Limited.

Furthermore, we use `fix` to make recursive definitions and the angle brackets $\langle$ and $\rangle$ to construct pairs. The constructs `isnil` *e*, `hd` *e*, `tl` *e*, `fix` *e* and `if` e_1 `then` e_2 `else` e_3 are built-in constructs of the language, and thus we omit the explicit parentheses. Additionally, we have used the (ordinary) parentheses (and) to indicate the parsing of the expression.

3 Types made explicit

Both Miranda and Standard ML enjoy the property that a programmer need not specify the types of the entities defined in the program. The implementations of the languages are able to infer those types if the program can be consistently typed at all. This is important for the functional programming style because then the higher-order functions can be instantiated much more freely.

As an example, implementations of Miranda and Standard ML will infer that the type of `reduce` is

$$(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\ \texttt{list} \rightarrow \beta$$

where α and β are type variables. The occurrence of `reduce` in the definition of `sum` has the type

$$(\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}) \rightarrow \texttt{Int} \rightarrow \texttt{Int list} \rightarrow \texttt{Int}$$

because it is applied to arguments of type `Int`$\rightarrow$`Int`$\rightarrow$`Int` and `Int`.

The enriched λ-calculus is equipped with a type inference system closely related to that found in Miranda and Standard ML (Milner, 1978; Damas, 1985), but somewhat more monomorphic. As an example, the function application *e*(*e*') is only well-formed if the type of *e* has the form $t \rightarrow t'$ and if the type of *e*' is *t*, and then the type of the application is *t*'. Another example is the construct `isnil` *e*, which is well-formed of type `Bool` provided that *e* is well-formed and has a type of the form *t* `list`. Similar rules exist for the other composite constructs of the language. For the constants we have axioms stating, e.g., that + has type `Int`$\times$`Int`$\rightarrow$`Int`, and that 0 has type `Int`. Based upon such axioms and rules we can infer that the program `sum` is well-formed, and we can determine the types of the various subexpressions. For some programs these types may contain type variables, and these type variables will then be replaced by a trivial type called `Void`. The purpose of the type following the keyword `HAS` is to avoid all type variables to be replaced by `Void`: those type variables that must match a subtype of the type following `HAS` will be instantiated accordingly.

The next step is to *annotate* the program with the inferred type information: we shall add the actual types to the constants and the bound variables of λ-abstractions. This means that the type analysis will transform the untyped program for `sum` (of Section 2) into the following typed program (to be called $\texttt{sum}_t$)

```
DEF reduce = λf[Int→Int→Int].λu[Int].
                fix (λg[Int list→Int]. λxs[Int list].
                        if isnil xs then u
                        else f (hd xs) (g(tl xs)))
VAL reduce (λx[Int].λy[Int]. +[Int×Int→Int](⟨x,y⟩))
                                                    (0[Int])
HAS Int list→Int
```

Note that the type of `reduce` is fixed. This is possible because there is only *one* application of `reduce` in the program. Otherwise, we may have to duplicate the definition, as the development to be performed does not (yet) allow the use of type variables. (This is what was meant above by the type system being more monomorphic than those for Miranda and Standard ML.)

4 Binding time made explicit

Neither Miranda, Standard ML nor the enriched λ-calculus has an explicit distinction between binding times. However, for higher-order functions we can distinguish between the parameters that are known and those that are not. The idea is now to capture this implicit distinction between binding times, and then annotate the operations of the enriched λ-calculus accordingly.

4.1 2-level syntax

We use the types of functions to record when their parameters will be available and their results produced. For `sum` it is clear that the list parameter is not available at compile-time, and we record this by underlining the corresponding component of the type

$$\mathtt{Sum}_t = \underline{\mathtt{Int}}\ \underline{\mathtt{list}} \rightarrow \mathtt{Int}$$

The fact that the list argument is not available at compile-time will have consequences for when the parameters are available for `reduce`. Again, we record this by underlining parts of the type

$$\mathtt{Reduce}_t = (\mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \mathtt{Int}) \rightarrow \mathtt{Int} \rightarrow \underline{\mathtt{Int}}\ \underline{\mathtt{list}} \rightarrow \mathtt{Int}$$

$\mathtt{Sum}_t$ and $\mathtt{Reduce}_t$ are examples of *2-level types*.

We interpret an underlined type as meaning that values of that type cannot be expected to be available until run-time. On the other hand, a type that is not completely underlined will denote values that definitely will be available at compile-time. With this intuition in mind, we claim that $\mathtt{Sum}_t$ and $\mathtt{Reduce}_t$ are unacceptable annotations because they denote functions that operate on actual run-time data, but at compile-time. Our point of view shall therefore be that $\mathtt{Sum}_t$ and $\mathtt{Reduce}_t$ are *incomplete* annotations, and that we need to make them well-formed.

To motivate our definition of a *well-formed* 2-level type we need to take a closer look at the interplay between the compile-time level and the run-time level. Thinking in terms of a compiler, it is quite clear that at compile-time we can manipulate pieces of code (to be executed at run-time), but we cannot manipulate entities computed at run-time. Hence, at compile-time we cannot directly manipulate objects of type $\underline{\mathtt{Int}}\ \underline{\mathtt{list}}$, whereas we can manipulate objects of type $\underline{\mathtt{Int}}\ \underline{\mathtt{list}} \mathbin{\underline{\rightarrow}} \underline{\mathtt{Int}}$, because the latter type may be regarded as the type of code for functions (to be executed at run-time). So, $\mathtt{Int} \rightarrow \underline{\mathtt{Int}}\ \underline{\mathtt{list}} \mathbin{\underline{\rightarrow}} \underline{\mathtt{Int}}$ will be the type of a function that, given an integer at compile-time, will produce a piece of code that has to be executed at run-time. Similarly, $\underline{\mathtt{Int}} \mathbin{\underline{\rightarrow}} \underline{\mathtt{Int}}\ \underline{\mathtt{list}} \mathbin{\underline{\rightarrow}} \underline{\mathtt{Int}}$ will be the type of a function to be executed at

$$\begin{array}{ll}
\mathtt{Reduce}_1 = & (\mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \mathtt{Int}) \rightarrow \mathtt{Int} \rightarrow \mathtt{Int}\ \mathtt{list} \rightarrow \mathtt{Int} \\
\mathtt{Reduce}_2 = & (\mathtt{Int} \rightarrow \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}) \rightarrow \mathtt{Int} \rightarrow \mathtt{Int}\ \mathtt{list} \rightarrow \mathtt{Int} \\
\mathtt{Reduce}_3 = & (\mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \mathtt{Int}) \rightarrow \mathtt{Int} \rightarrow \underline{\mathtt{Int}}\ \underline{\mathtt{list}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}} \\
\mathtt{Reduce}_4 = & (\underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}) \rightarrow \mathtt{Int} \rightarrow \mathtt{Int}\ \mathtt{list} \rightarrow \mathtt{Int} \\
\mathtt{Reduce}_5 = & (\mathtt{Int} \rightarrow \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}) \rightarrow \mathtt{Int} \rightarrow \underline{\mathtt{Int}}\ \underline{\mathtt{list}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}} \\
\mathtt{Reduce}_6 = & (\mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \mathtt{Int}) \rightarrow \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}\ \underline{\mathtt{list}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}} \\
\mathtt{Reduce}_7 = & (\underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}) \rightarrow \mathtt{Int} \rightarrow \underline{\mathtt{Int}}\ \underline{\mathtt{list}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}} \\
\mathtt{Reduce}_8 = & (\mathtt{Int} \rightarrow \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}) \rightarrow \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}\ \underline{\mathtt{list}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}} \\
\mathtt{Reduce}_9 = & (\underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}) \rightarrow \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}\ \underline{\mathtt{list}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}} \\
\mathtt{Reduce}_{10} = & (\underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}})\ \underline{\rightarrow}\ \underline{\mathtt{Int}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}\ \underline{\mathtt{list}}\ \underline{\rightarrow}\ \underline{\mathtt{Int}}
\end{array}$$

Fig. 1. Well-formed 2-level types for `reduce`.

run-time, whereas `Int`→`Int list`→`Int` will be the type of a function to be executed at compile-time.

To be a bit more precise, we shall say that an 'all underlined' function type is well-formed, and we shall call it a *run-time type*. A type that does not contain any underlinings will also be well-formed, since it will be the type of values fully computed at compile-time. Well-formed types can then be combined using the type constructors →, × and `list`.

Corresponding to the type of `sum` we have two well-formed 2-level types

$$\mathtt{Sum}_1 = \mathtt{Int}\ \mathtt{list} \rightarrow \mathtt{Int}$$

$$\mathtt{Sum}_2 = \underline{\mathtt{Int}}\ \underline{\mathtt{list}}\,\underline{\rightarrow}\,\underline{\mathtt{Int}}$$

For `reduce` we have the ten well-formed 2-level types shown in fig. 1.

We shall say that two 2-level types have the same *underlying type* if they are equal

$$\begin{tikzcd}
 & & \mathtt{Reduce}_1 \arrow[dl, dash] \arrow[dr, dash] & & \mathtt{Reduce}_t \arrow[dl, dash] \\
 & \mathtt{Reduce}_2 \arrow[dl, dash] \arrow[dr, dash] & & \mathtt{Reduce}_3 \arrow[dl, dash] \arrow[dr, dash] & \\
\mathtt{Reduce}_4 \arrow[dr, dash] & & \mathtt{Reduce}_5 \arrow[dl, dash] \arrow[dr, dash] & & \mathtt{Reduce}_6 \arrow[dl, dash] \\
 & \mathtt{Reduce}_7 \arrow[dr, dash] & & \mathtt{Reduce}_8 \arrow[dl, dash] & \\
 & & \mathtt{Reduce}_9 \arrow[d, dash] & & \\
 & & \mathtt{Reduce}_{10} & &
\end{tikzcd}$$

Fig. 2. Compatible 2-level types for `reduce`.

except for the underlinings. One 2-level type is then *compatible* with another if they have the same underlying type, and if underlined occurrences in the latter are also underlined in the former. Thus the intention is that compatible types may be obtained by 'moving' data, computations or results from compile-time to run-time, but not *vice versa*. As an example, $\texttt{Reduce}_3$ is compatible with $\texttt{Reduce}_t$, but $\texttt{Reduce}_2$ is not. This is expressed by the Hasse diagram of fig. 2.

We shall say that the *best completion* of $\texttt{Reduce}_t$ is $\texttt{Reduce}_3$, because all other well-formed 2-level types that are compatible with $\texttt{reduce}_t$ also will be compatible with $\texttt{Reduce}_3$. (Shortly we shall see that the actual definition of `reduce` will impose further restrictions upon the type.) Similarly, the best completion of $\texttt{Sum}_t$ is $\texttt{Sum}_2$. A formal account may be found in Nielson and Nielson (1988*a*).

4.2 Binding time analysis

We now annotate complete programs so that it becomes explicit which computations should be performed at compile-time (namely, those that are not underlined), and which should be postponed until run-time (namely, those that are underlined). The resulting program is called a *2-level program* and similarly, an annotated expression is called a *2-level expression*.

The above discussion suggests that the annotation of `reduce` should reflect the binding time information given by $\texttt{Reduce}_3$. However, it is not possible to annotate the definition of `reduce` so that it is both well-formed and will have type $\texttt{Reduce}_3$. The reason is that the binding times do *not* match: according to $\texttt{Reduce}_3$, the function argument `f` has type `Int→Int→Int`, and the list argument `xs` has type $\underline{\texttt{Int}}\ \underline{\texttt{list}}$, so `f(hd xs)` will supply `f` with a run-time argument even though it expects a compile-time argument. This will be formalized by defining a *well-formedness condition* on 2-level expressions (Nielson and Nielson , 1988*b*)*. There are two ingredients in this: one is that the underlying expression (obtained by removing the underlinings) must type-check properly; the other is that the binding times must agree. Thus a function with a run-time type must be given a run-time argument and we express this by underlining the λ-abstraction of the function definition, as in $\underline{\lambda}$xs[]. ..., and by underlining the parentheses in the function application, as in f $\underline{(}\ldots\underline{)}$. Similarly, a function with a compile-time type must be supplied with a compile-time argument, and in that case we neither underline the λ-abstraction nor the parentheses of the function application.

Given an incomplete annotation $\texttt{sum}_t$ of `sum`

```
DEF reduce = λf[Int→Int→Int].λu[Int].
                fix (λg[Int list→Int]. λxs[Int list].
                    if isnil xs then u
                    else f(hd xs) (g(tl xs)))
VAL reduce (λx[Int].λy[Int]. +[Int×Int→Int] (⟨x,y⟩))
                                                (0[Int])
HAS Int list→Int
```

* Alternative well-formedness conditions are studied by Nielson (1988), Nielson and Nielson (1988*a*, 1990) and Schmidt (1988).

the *binding time analysis* (Nielson, 1988; Nielson and Nielson, 1988*b*; Schmidt, 1988) will complete the annotation so that the resulting 2-level program is

- well-formed (as explained above);
- postpones as few computations as possible to run-time; and
- is compatible with the incomplete annotation (where compatibility for expressions is very much as for types).

The program obtained[*] from $\texttt{sum}_t$ is $\texttt{sum}_9$

$$
\begin{array}{l}
\texttt{DEF reduce}_9 = \lambda\texttt{f[}\underline{\texttt{Int}\to\texttt{Int}\to\texttt{Int}}\texttt{]}.\underline{\lambda}\texttt{u[}\underline{\texttt{Int}}\texttt{]}. \\
\qquad\qquad\quad \underline{\texttt{fix}}\ (\underline{\lambda}\texttt{g[}\underline{\texttt{Int}}\ \underline{\texttt{list}}\to\underline{\texttt{Int}}\texttt{]}.\ \underline{\lambda}\texttt{xs[}\underline{\texttt{Int}}\ \underline{\texttt{list}}\texttt{]}. \\
\qquad\qquad\qquad\quad \underline{\texttt{if}}\ \underline{\texttt{isnil}}\ \texttt{xs}\ \underline{\texttt{then}}\ \texttt{u} \\
\qquad\qquad\qquad\quad \underline{\texttt{else}}\ \texttt{f}\underline{\texttt{(hd}}\ \texttt{xs}\underline{\texttt{)(}}\texttt{g}\underline{\texttt{(tl}}\ \texttt{xs}\underline{\texttt{))}}\texttt{)} \\
\texttt{VAL reduce}_9\ (\underline{\lambda}\texttt{x[}\underline{\texttt{Int}}\texttt{]}.\underline{\lambda}\texttt{y[}\underline{\texttt{Int}}\texttt{]}.+\texttt{[}\underline{\texttt{Int}\times\texttt{Int}\to\texttt{Int}}\texttt{]}\underline{(\langle}\texttt{x, y}\underline{\rangle)}) \\
\hfill \underline{(}\texttt{0[}\underline{\texttt{Int}}\texttt{]}\underline{)} \\
\texttt{HAS}\ \underline{\texttt{Int}}\ \underline{\texttt{list}\to\texttt{Int}}
\end{array}
$$

where $\texttt{reduce}_9$ has type $\texttt{Reduce}_9$. Note that we cannot use $\lambda\texttt{u[}\underline{\texttt{Int}}\texttt{]}\ldots.$ instead of $\underline{\lambda}\texttt{u[}\underline{\texttt{Int}}\texttt{]}\ldots.$ as the well-formedness conditions (Nielson and Nielson, 1988*b*) on types only allow us to manipulate run-time *function* types at compile-time, and not run-time data types like $\underline{\texttt{Int}}$.

The remaining well-formed 2-level programs are less interesting. Corresponding to the 2-level type $\texttt{Reduce}_1$ we have a program with no underlinings at all, and corresponding to $\texttt{Reduce}_{10}$ we have a program where all operations are underlined. These are the only well-formed 2-level programs with `sum` as the underlying program[†]. Note that in each of these programs the binding times for `reduce` is fixed, and this is possible because there is only *one* application of `reduce` in the program. Otherwise, we may have to duplicate the definition of the function.

4.3 Improving the binding time analysis

The annotation obtained from the binding time analysis is *optimal* in the sense that as few computations as possible are postponed until run-time. However, it is often the case that a slight rewriting of the program will produce an even better distinction between the binding times. As an example, the order of the parameters of a function may be changed or the representation of data types may be modified. To illustrate this consider the function `lookup` of type

$$(\texttt{Name}\times\underline{\texttt{Int}})\ \texttt{list}\to\texttt{Name}\to\texttt{Int}$$

where we assume that the second components of all pairs in the first argument are unknown at compile-time. We then have a situation where known and unknown data

[*] Actually, we apply an adapted version of the binding time analysis algorithm of Nielson and Nielson (1988*a*) in order to comply with the well-formedness predicate of Nielson and Nielson (1988*b*).

[†] This statement holds for the well-formedness predicate of Nielson and Nielson (1988*b*). If the predicate of Nielson (1988) and Nielson and Nielson (1988*a*) is used, then there will be two more well-formed programs.

are mixed, and the binding time analysis of Nielson and Nielson (1988*a*) will return a function with the annotated type

$$(\underline{\texttt{Name}\times\texttt{Int}})\ \underline{\texttt{list}}\,\underline{\to}\,\underline{\texttt{Name}}\,\underline{\to}\,\underline{\texttt{Int}}$$

so that all computations will be postponed until run-time. However, we may split the list of pairs into two lists and rearrange the order of the parameters so that the type becomes

$$\texttt{Name list}\to\texttt{Name}\to\underline{\texttt{Int}}\ \texttt{list}\to\texttt{Int}$$

We then get a much better distinction between the binding times, because now only the elements of the *second* list will be unknown at compile-time, and the binding time analysis will return a function with the annotated type

$$\texttt{Name list}\to\texttt{Name}\to\underline{\texttt{Int}}\ \underline{\texttt{list}}\,\underline{\to}\,\underline{\texttt{Int}}$$

Hence, some of the computations can be performed at compile-time, and this idea is further explored in Nielson and Nielson (1990*b*).

The above example is rather involved in that it changes the overall type of the function. However, it may also be useful to perform some more local transformations. Consider the program $\texttt{sum}_9$ of the previous subsection. Here the fixed point operation of $\texttt{reduce}_9$ is underlined, and intuitively this means that we cannot use the recursive structure of its body at compile-time, e.g., during abstract interpretation and when generating code. We may therefore want to replace the run-time fixed point by a compile-time fixed point. Since $\texttt{sum}_9$ is the best completion of $\texttt{sum}_t$ we cannot obtain this effect by simply changing the annotation. The idea is therefore first

- to *transform* the underlying program; and then
- to *repeat* the binding time analysis.

In our case we shall apply the transformation

$$\lambda x[t_1].\,\texttt{fix}(\lambda f[t_2\to t_3].e)\Rightarrow\texttt{fix}(\lambda g[t_1\to t_2\to t_3].\lambda x[t_1].e[g(x)/f])$$

that replaces the first pattern with the second. Here $e[g(x)/f]$ is e with all occurrences of f replaced by $g(x)$, and g is assumed to be a fresh identifier. We then repeat the binding time analysis with $\texttt{Sum}_t$ as the 'goal' type and get

$$\begin{array}{l}
\texttt{DEF reduce}_{9a}=\lambda\texttt{f[}\underline{\texttt{Int}\to\texttt{Int}\to\texttt{Int}}\texttt{].}\\
\qquad\qquad\texttt{fix (}\lambda\texttt{g'[}\underline{\texttt{Int}}\,\underline{\to}\,\underline{\texttt{Int}}\ \underline{\texttt{list}}\texttt{].}\ \underline{\lambda}\texttt{u[}\underline{\texttt{Int}}\texttt{].}\ \underline{\lambda}\texttt{xs[}\underline{\texttt{Int}}\ \underline{\texttt{list}}\texttt{].}\\
\qquad\qquad\qquad\quad\underline{\texttt{if}}\ \underline{\texttt{isnil}}\ \texttt{xs}\ \underline{\texttt{then}}\ \texttt{u}\\
\qquad\qquad\qquad\quad\underline{\texttt{else}}\ \texttt{f}\underline{\texttt{(hd}}\ \texttt{xs}\underline{\texttt{)}}\ \underline{\texttt{(}}\texttt{g'}\,\underline{\texttt{(}}\texttt{u}\underline{\texttt{)}}\ \underline{\texttt{(tl}}\ \texttt{xs}\underline{\texttt{))}}\texttt{)}\\
\texttt{VAL reduce}_{9b}\ \texttt{(}\underline{\lambda}\texttt{x[}\underline{\texttt{Int}}\texttt{].}\underline{\lambda}\texttt{y[}\underline{\texttt{Int}}\texttt{].}\ +\texttt{[}\underline{\texttt{Int}\times\texttt{Int}}\,\underline{\to}\,\underline{\texttt{Int}}\texttt{]}\ \underline{\texttt{(}\langle}\texttt{x, y}\underline{\rangle\texttt{)}}\texttt{)}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\underline{\texttt{(}}\texttt{0[}\underline{\texttt{Int}}\texttt{]}\underline{\texttt{)}}\\
\texttt{HAS}\ \underline{\texttt{Int}}\ \underline{\texttt{list}}\,\underline{\to}\,\underline{\texttt{Int}}
\end{array}$$

where the fixed point now is computed at compile-time and the function g' is bound at compile-time.

As a side-effect the functionality of the fixed point has been changed and, in particular, g' has become a higher-order function. For some stack-based implementations it is expensive to handle higher-order functions, and we may therefore want to transform the program further to change the functionality of g'. To do that we first apply the following transformation to the underlying program of $\texttt{sum}_{9a}$

$$\texttt{fix}(\lambda f[t_1 \to t_2 \to t_3].\lambda x[t_1].\lambda y[t_2].e) \Rightarrow$$
$$\lambda x[t_1].\lambda y[t_2].(\texttt{fix}(\lambda g[t_1 \times t_2 \to t_3].\lambda z[t_1 \times t_2].$$
$$e[\lambda x[t_1].\lambda y[t_2].g(\langle x,y\rangle)/f][\texttt{fst}\,z/x][\texttt{snd}\,z/y]))$$
$$(\langle x,y\rangle)$$

Here g and z are assumed to be fresh identifiers. Next we apply the β-transformation

$$(\lambda x[t].e)(e') \Rightarrow e[e'/x]$$

(recalling that our semantics is non-strict). The binding time analysis is then applied to the resulting program with $\texttt{Sum}_t$ as the overall annotated type and we get the program $\texttt{sum}_{9b}$

```
DEF reduce9b = λf[Int→Int→Int]. λu[Int].λxs[Int list].
                (fix (λg[Int × Int list→Int].
                      λz[Int × Int list].
                        if isnil (snd z) then fst z
                        else f (hd (snd z))
                (g (⟨fst z, tl (snd z)⟩))))
                (⟨u,xs⟩)
VAL reduce9b (λx[Int].λy[Int]. +[Int×Int→Int](⟨x,y⟩))
                                                    (0[Int])
HAS Int list→Int
```

Note, however, that the motivation for this transformation is not that fewer computations are postponed until run-time. Rather, the transformation will allow us to specify a backward abstract interpretation in Section 8. Also note that the two transformations preserve total correctness.

4.4 Remarks

We have experimented with different ways in which the two binding levels may interact (Nielson, 1988; Nielson and Nielson, 1988*a*, *b*), but the restrictions have always been rather conservative. As an example, we have not allowed values computed at compile-time to be used directly at run-time because of the computational consequences of allowing this for function types. Also, our binding time analysis does not change the underlying program. Flexibility is introduced by allowing arbitrary (semantics-preserving) program transformations, e.g., expressed as simple pattern matching templates, or as in Nielson and Nielson (1990*b*) within the unfold/fold framework of Burstall and Darlington (1977).

There are alternative approaches to binding time analysis that allow greater interaction between the levels, and which incorporate certain program transformations (e.g., see Jones *et al.*, 1985; Jørring and Scherlis, 1986; Montenyohl and Ward, 1988). The price to be paid is that it becomes harder to reason about the correctness properties. Our approach gives a better separation of concerns: it is fairly straightforward to prove the correctness and optimality of our binding time analysis algorithm, and at the same time we are not restricted in the kind of program transformations that can be made (Nielson and Nielson, 1988 *b*).

Our approach to binding time analysis is purely *syntactic* whereas, e.g., Mogensen (1989) takes a more semantic approach using projections. In fact, our approach is strongly motivated by the (syntactic) approach to polymorphic type inference of Milner (1975). The structure of our binding time analysis algorithm is very close to Milner's (1975) algorithm $\mathcal{W}$, the main difference being that we have no counterpart of substitutions, and therefore have to perform some extra recursive calls to 'unify' the binding time annotations.

Finally, we should point out that we only allow explicit monotypes in Section 3 because it is unclear how to do binding time annotations for type variables. In particular, the identity function λx. x might have type $\alpha \underline{\rightarrow} \alpha$ and $\alpha \rightarrow \alpha$, depending upon the kind of 2-level types that α may range over.

5 Combinators made explicit

The binding time information of a 2-level program clearly indicates which computations should be carried out at compile-time and which should be carried out at run-time. The compile-time computations should be executed by a compiler, and it is well-known how to do this. The run-time computations should give rise to code instead. We may also want to perform some data flow analyses either in order to validate some program transformations or to improve the efficiency of the code generated. It is important to observe that it is the run-time computations, not the compile-time computations, that should be analysed, just as it is the run-time computations, not the compile-time computations, that should give rise to code. This then calls for the ability to *interpret the run-time constructs* in different ways, depending upon the task at hand.

It is not straightforward to do so when the run-time computations are expressed in the form of λ-expressions. As an example, the usual meaning of

$$\underline{\lambda}\mathtt{x}[\underline{\mathtt{Int}\times\mathtt{Int}}].\ \mathtt{f}\ \underline{(}\mathtt{g}\ \underline{(}\mathtt{x}\underline{)}\underline{)}$$

is $\lambda v.f(g\,v)$. However, we may be interested in an analysis determining whether both components of x are needed in order to compute the result. This is an example of a backward data flow analysis and, as we shall see later, the natural interpretation of the expression will then be $\lambda v.g(f\,v)$. It is not straightforward to interpret function abstraction and function application so as to be able to obtain both meanings. The idea is therefore to focus on functions and functionals (expressed as combinators, as in Backus, 1978; and Curien, 1986) rather than values and functions. We then write

$$\mathtt{f}\ \square\ \mathtt{g}$$

for the expression above, and the effect of both $\lambda v.f(g\,v)$ and $\lambda v.g(f\,v)$ can be obtained by suitably reinterpreting the functional $\square$.

This observation calls for transforming the run-time computations into combinator form. Similar considerations motivate the use of combinators in the implementation of functional languages (Turner, 1979; Cousineau *et al.*, 1987; Hughes, 1982), and the use of categorical combinators when interpreting the typed λ-calculus in an arbitrary cartesian closed category (Lambeck and Scott, 1986). However, it is important to stress that we shall leave the compile-time computations in the form of λ-expressions, and only transform the run-time computations into combinator form.

5.1 Combinator introduction

A 2-level program (or 2-level expression) is in *combinator form* whenever all the run-time computations are expressed as *categorical combinators*. For the program `sum`$_{9b}$ the corresponding program `sum`$_{9B}$ in combinator form will be

```
DEF reduce_9B = λf[ ].Curry (fix(λg[ ].
                    Cond(Isnil[ ]□Snd[ ], Fst[ ],
                         Apply[ ]□Tuple(f□Hd[ ]□Snd[ ],
                           g□Tuple(Fst[ ], Tl[ ]□Snd[ ])))))
VAL Apply[ ]□Tuple((reduce_9B (Curry +[ ]))□(Const[ ]0[ ]),
                         Id[ ])
HAS Int list→Int
```

as we will explain below. For the sake of readability we have omitted the type information (in square brackets).

The conditional of `reduce`$_{9B}$ has been replaced by the combinator `Cond` where the intention is that

$$\texttt{Cond}(f,g,h) = \underline{\lambda} x.\ \underline{\texttt{if}}\ f\underline{(}x\underline{)}\ \underline{\texttt{then}}\ g\underline{(}x\underline{)}\ \underline{\texttt{else}}\ h\underline{(}x\underline{)}$$

Within the conditional the test $\underline{\texttt{isnil}}$ ($\underline{\texttt{snd}}$ z) is replaced by `Isnil[]`□`Snd[]` because the run-time parameter will be implicit. Similarly, the true-branch $\underline{\texttt{fst}}$ z is replaced by `Fst[]`. Here `Isnil`, `Fst` and `Snd` are combinators corresponding to $\underline{\texttt{isnil}}$ *e*, $\underline{\texttt{fst}}$ *e* and $\underline{\texttt{snd}}$ *e*, respectively. To understand the transformation of the false-branch, note that the intention with the combinators `Apply` and `Tuple` is given by

$$\texttt{Apply} = \underline{\lambda\langle} f, x\underline{\rangle}.f\underline{(}x\underline{)}$$

$$\texttt{Tuple}(f,g) = \underline{\lambda} x.\underline{\langle} f\underline{(}x\underline{)}, g\underline{(}x\underline{)}\underline{\rangle}$$

and that `Hd` and `Tl` correspond to $\underline{\texttt{hd}}$ *e* and $\underline{\texttt{tl}}$ *e*, respectively.

The function `reduce` takes its arguments one at a time, and this is achieved using the combinator `Curry`

$$\texttt{Curry}f = \underline{\lambda} x.\underline{\lambda} y.f\underline{(}\underline{\langle} x,y\underline{\rangle}\underline{)}$$

In the expression part of the program we use the additional combinators `Id` and `Const`, and the intention is that `Id` is the identity function and `Const`f equals $\underline{\lambda} x.f$.

To transform well-formed 2-level programs into combinator form we may use the algorithm of Nielson and Nielson (1988*b*)*. It is an extension of the usual algorithm for translating the typed λ-calculus into categorical combinators (Curien, 1986). Thus for each of the programs $\texttt{sum}_1$, $\texttt{sum}_9$, $\texttt{sum}_{9a}$, $\texttt{sum}_{9b}$ and $\texttt{sum}_{10}$ we will have equivalent programs in combinator form. In the remainder of this paper we shall restrict our attention to $\texttt{sum}_{9B}$.

5.2 Partial evaluation

The program $\texttt{sum}_{9B}$ can be further simplified because we can supply the occurrence of `reduce` with its first parameter which is known at compile-time. For this we need the two transformations

$$\texttt{DEF}\, f = e\, \texttt{VAL}\, e'\, \texttt{HAS}\, t \Rightarrow \texttt{VAL}\, e'[e/f]\, \texttt{HAS}\, t$$

$$(\lambda x.e')\,(e) \Rightarrow e'[e/x]$$

which preserve total correctness (due to the use of a non-strict semantics). This sort of program transformation is often called *partial evaluation* (Ershov, 1982; Jones *et al.*, 1985; Nielson, 1988), and in our case it gives rise to the program

```
VAL Apply[ ] □ Tuple((Curry (fix(λg[ ].
    Cond(Isnil[ ] □ Snd[ ], Fst[ ],
         Apply[ ] □ Tuple((Curry +[ ]) □ Hd[ ] □ Snd[ ],
           g □ Tuple(Fst[ ], Tl[ ] □ Snd[ ])))))
      □ (Const[ ] O[ ]), Id[ ])
HAS Int list→Int
```

This transformation could equally well have been performed before the combinators were introduced.

5.3 Algebraic transformation

The run-time counterpart of partial evaluation is called *algebraic transformation* (Backus, 1978). An example is the transformation

$$\texttt{Apply[]} \,\square\, \texttt{Tuple((Curry}\, e) \,\square\, e', e'') \Rightarrow e \,\square\, \texttt{Tuple}(e', e'')$$

If we apply this transformation twice to the above program then we get the program $\texttt{sum}'_{9B}$

```
VAL fix (λg[ ].Cond(Isnil[ ] □ Snd[ ], Fst[ ],
                    +[ ] □ Tuple(Hd[ ] □ Snd[ ],
                    g □ Tuple(Fst[ ], Tl[ ] □ Snd[ ]))))
  □ Tuple(Const[ ] O[ ], Id[ ])
HAS Int list→Int
```

* The algorithm in Nielson and Nielson (1990*a*) may also be used if the more general well-formedness predicate of Nielson and Nielson (1988*a*, *b*) is used (in Section 4), but unfortunately it uses a rather unnatural combinator in certain cases.

Note that by now all higher-order run-time functions have disappeared. Also, note that the transformation preserves total correctness.

The algebraic transformations play an important role in simplifying programs. The algorithm for combinator introduction proceeds by structural induction, and the algebraic transformations can be used to reduce certain unnecessarily complicated expressions that arise in this process. Also, the application of partial evaluation and algebraic transformations can be intertwined. In Section 8 we shall see how the algebraic transformations can simplify results obtained from further transformations.

5.4 Remarks

We have used the categorical combinators for mainly two reasons. One is that they are *functionally complete*, meaning that their expressive power is equivalent to that of the typed λ-calculus (Curien, 1986). As a consequence of this it is possible to translate expressions in the typed λ-calculus into expressions of the categorical combinatory logic. The other reason is that we get a *fixed set of combinators*, and this is important in the next section where we introduce the concept of a parameterized semantics.

We believe that the development of this section can be performed for any fixed set of combinators that is sufficiently expressive. The traditional algorithms for combinator introduction must then be extended so that the combinators are introduced only for the run-time constructs of the notation. The concept of partial evaluation is unchanged as it does not depend on the run-time level at all, whereas the algebraic transformations must be modified so as to use algebraic laws that hold for the actual set of combinators.

Some of the most popular approaches to implementing functional language use supercombinators (Hughes, 1982; Peyton Jones, 1987). The actual set of combinators will then depend upon the program at hand so that our approach does not directly carry over. It should be fairly straightforward to extend the algorithm for introducing supercombinators (Hughes, 1982) so that supercombinators are generated only for the run-time constructs. The concept of algebraic transformation will have to be modified, since we do not have algebraic laws in the same sense as before. For this one may explore some variant of partial evaluation that only considers how to rewrite the supercombinators.

6 Parameterized semantics

We want to interpret the run-time constructs in different ways, depending on the task at hand, and at the same time we want the meaning of the compile-time constructs to be fixed. To make this possible we shall *parameterize* the semantics on an *interpretation* (Nielson and Nielson, 1986, 1988; Nielson, 1987, 1989) specifying the meaning of the run-time level. The interpretation will define

- the meaning of the run-time function types; and
- the meaning of the combinators (and the compile-time fixed point).

Relative to an interpretation one can then define the semantics of all well-formed

2-level types, of all well-formed 2-level programs and of all well-formed 2-level expressions in combinator form. In the next sections we give some example interpretations that specify forward and backward data flow analyses and code generation. The correctness of these interpretations will be expressed relative to a *non-strict* (i.e., lazy) standard interpretation, to be given below.

6.1 The type part of an interpretation

The *type part* of an interpretation $\mathcal{I}$ will specify a set $\mathcal{I}_{t \underline{\rightarrow} t'}$ of values for each well-formed type $t \underline{\rightarrow} t'$. (Technically, this set is extended with a partial ordering expressing when one value is less defined than another, but we shall dispense with this in the present paper.) For the standard interpretation $\mathcal{S}$ we have

$$\mathcal{S}_{t \underline{\rightarrow} t'} = [\![t]\!](\mathcal{S}) \rightarrow [\![t']\!](\mathcal{S})$$

where $[\![t]\!](\mathcal{S})$ and $[\![t']\!](\mathcal{S})$ are the sets of values of type t and t', respectively, and $\rightarrow$ constructs the appropriate function space. We can then define

$$[\![\underline{\texttt{Int}}]\!](\mathcal{S}) = \mathbf{Z} \quad \text{(the set of integers)}$$

$$[\![\underline{\texttt{Bool}}]\!](\mathcal{S}) = \mathbf{T} \quad \text{(the set of truth values)}$$

and more interestingly

$$[\![t \underline{\rightarrow} t']\!](\mathcal{S}) = [\![t]\!](\mathcal{S}) \rightarrow [\![t']\!](\mathcal{S}) \quad \text{(functions)}$$

$$[\![t \underline{\times} t']\!](\mathcal{S}) = [\![t]\!](\mathcal{S}) \times [\![t']\!](\mathcal{S}) \quad \text{(pairs of values)}$$

$$[\![t\ \underline{\texttt{list}}]\!](\mathcal{S}) = [\![t]\!](\mathcal{S})^* \quad \text{(sequences of values)}$$

so that $[\![t]\!](\mathcal{S})$ is defined for all run-time types.

Given the meaning of the run-time function types it is straightforward to extend it to all well-formed compile-time types. In general, the set $[\![t]\!](\mathcal{I})$ of values associated with the type t will be defined by structural induction, and we have

$$[\![\texttt{Int}]\!](\mathcal{I}) = \mathbf{Z}$$

$$[\![\texttt{Bool}]\!](\mathcal{I}) = \mathbf{T}$$

$$[\![t \rightarrow t']\!](\mathcal{I}) = [\![t]\!](\mathcal{I}) \rightarrow [\![t']\!](\mathcal{I})$$

$$[\![t \times t']\!](\mathcal{I}) = [\![t]\!](\mathcal{I}) \times [\![t']\!](\mathcal{I})$$

$$[\![t\ \texttt{list}]\!](\mathcal{I}) = [\![t]\!](\mathcal{I})^*$$

$$[\![t \underline{\rightarrow} t']\!](\mathcal{I}) = \mathcal{I}_{t \underline{\rightarrow} t'}$$

As an example, $[\![\texttt{Sum}_1]\!](\mathcal{S}) = [\![\texttt{Sum}_2]\!](\mathcal{S}) = \mathbf{Z}^* \rightarrow \mathbf{Z}$ so that the distinction between the compile-time function type and the run-time function type disappears (as must be expected in a standard semantics).

6.2 The expression part of an interpretation

The *expression part* of an interpretation specifies a value or a function for each of the combinators $\square$, `Tuple`, `Fst`, etc. In Fig. 3 we give parts of the specification of $\mathscr{S}$ using an appropriate mathematical notation (shown in boldface). The meaning of the compile-time constructs is predetermined except for that of `fix` e. Here the interpretation $\mathscr{I}$ specifies a function $\mathscr{I}^{t}_{\texttt{fix}}$ for each type t of `fix` e. In fig. 3 we write **FIX** for the (least) fixed point operator.

Given the meanings of the combinators and of `fix` it is now straightforward to extend it to expressions and programs. For expressions we define a value $[\![e]\!](\mathscr{I})$ in $[\![t]\!](\mathscr{I})$ for every well-formed expression e of type t. To cater for the free variables we need an environment *env* that associates the variables with their values. We shall not give all the details here, but some of the more interesting clauses are

$$[\![\lambda x[t].e]\!](\mathscr{I})\,env = [\![e]\!](\mathscr{I})\,env[\mathbf{v}/x]$$

$$[\![e(e')]\!](\mathscr{I})\,env = ([\![e]\!](\mathscr{I})\,env)([\![e']\!](\mathscr{I})\,env)$$

$$[\![x]\!](\mathscr{I})\,env = env(x)$$

where $env[v/x]$ is as env except that x has the value v. The next example clauses illustrate the use of the interpretation $\mathscr{I}$

$$[\![e \,\square\, e']\!](\mathscr{I})\,env = \mathscr{I}_{\square}([\![e]\!](\mathscr{I})\,env, [\![e']\!](\mathscr{I})\,env)$$

$$[\![\texttt{Tuple}(e, e')]\!](\mathscr{I})\,env = \mathscr{I}_{\texttt{Tuple}}([\![e]\!](\mathscr{I})\,env, [\![e']\!](\mathscr{I})\,env)$$

$$[\![\texttt{Fst}[\,]]\!](\mathscr{I})\,env = \mathscr{I}_{\texttt{Fst}}$$

$$[\![\texttt{fix}\,e]\!](\mathscr{I})\,env = \mathscr{I}^{t}_{\texttt{fix}}([\![e]\!](\mathscr{I})) \quad \text{where } \texttt{fix}\ e \text{ has type } t$$

To summarize, $[\![e]\!](\mathscr{I})\,env$ is defined by structural induction on e, and in order to compose the results obtained for the subexpressions we will either consult the interpretation $\mathscr{I}$ (if the constructor is a combinator or `fix`), or we will use a standard approach.

$$\mathscr{S}_{\square}\ \mathbf{(f,g)\ v = f(g\ v)}$$
$$\mathscr{S}_{\texttt{Tuple}}\ \mathbf{(f,g)\ v = \langle f\ v, g\ v\rangle}$$
$$\mathscr{S}_{\texttt{Fst}}\ \mathbf{(u,v) = u}$$
$$\mathscr{S}_{\texttt{Snd}}\ \mathbf{(u,v) = v}$$
$$\mathscr{S}_{\texttt{Hd}}\ \mathbf{(v{:}l) = v}$$
$$\mathscr{S}_{\texttt{Tl}}\ \mathbf{(v{:}l) = l}$$
$$\mathscr{S}_{\texttt{Isnil}}\ \mathbf{l = (l{=}[\ ]) \rightarrow true \mid (l{=}v{:}l') \rightarrow false}$$
$$\mathscr{S}_{\texttt{Cond}}\ \mathbf{(f,g,h)\ v = (f\ v{=}true) \rightarrow g\ v \mid (f\ v{=}false) \rightarrow h\ v}$$
$$\mathscr{S}_{\texttt{Const}}\ \mathbf{f\ v = f}$$
$$\mathscr{S}_{\texttt{Id}}\ \mathbf{v = v}$$
$$\mathscr{S}_{+}\ \mathbf{\langle u,v\rangle = u+v}$$
$$\mathscr{S}^{t}_{\texttt{fix}}\ \mathbf{f = FIX\ f} \quad \text{for all } t$$

Fig. 3. Fragments of the expression part of $\mathscr{S}$.

Finally, the semantics of a program is defined to be that of the expression after 'VAL'. The specification of $\mathscr{S}$ given in fig. 3 is sufficient to determine that

$$[\![\mathtt{sum}'_{9B}]\!](\mathscr{S})\mathbf{l} = \mathbf{FIX}(\lambda \mathbf{g}.\lambda\langle \mathbf{v},\mathbf{l}\rangle.(\mathbf{l} = [\,]) \rightarrow \mathbf{v} \mid (\mathbf{l} = \mathbf{a}:\mathbf{l'}) \rightarrow \mathbf{a}+\mathbf{g}\langle \mathbf{v},\mathbf{l'}\rangle)\langle \mathbf{0},\mathbf{l}\rangle.$$

6.3 Remarks

In the previous sections we have presented type inference, binding time analysis and combinator introduction as purely syntactic manipulations of programs. We have only referred to (a vague and unspecified notion of) semantics when talking about program transformations, and the extent to which they are correctness preserving. Having introduced the parameterized semantics and the standard interpretation $\mathscr{S}$ we can now take at least two approaches.

One approach is to view the parameterized semantics as the ultimate semantics for the original enriched λ-calculus of Section 2. Taking sum as an example we thus have to

- transform it into an explicitly typed program (i.e. $\mathtt{sum}_t$);
- then into an explicitly annotated program without doing any program transformations (i.e., $\mathtt{sum}_9$ rather than $\mathtt{sum}_{9b}$); and
- finally into combinator form without doing any partial evaluation or algebraic transformations (much as $\mathtt{sum}_{9B}$ was obtained from $\mathtt{sum}_{9b}$).

The semantics of sum then amounts to the parameterized semantics of the resulting program, say $\mathtt{sum}_C$. We can then express the correctness of the program transformations, partial evaluations and algebraic transformations as the condition that they do not change the final semantics regardless of the interpretation considered.

Another approach is to give a direct definition of the semantics of the enriched λ-calculus. Then one must ensure that the semantics of an untyped program (e.g., sum) corresponds to the parameterized semantics with respect to $\mathscr{S}$ of the combinator version of the program (e.g., $\mathtt{sum}_C$). We shall not go deeply into this issue here, but we favour the first approach above, as our ultimate interest is in non-standard interpretations, e.g., interpretations for code generation.

The combination of using a denotational as well as a parameterized semantics facilitates the development of various *meta-theories*, i.e., frameworks for correctness proofs of abstract interpretations (Nielson, 1987, 1989) and code generation (Nielson and Nielson, 1988*c*). Also, note that the notion of a parameterized semantics would have been a very challenging task to define if we had not restricted our attention to a fixed set of combinators.

7 Forward abstract interpretation

In the previous sections we have seen some very simple program transformations that are *universally* applicable. This means that whenever the left-hand side pattern matches an expression one may replace the expression by the right-hand side pattern.

However, in other cases it is necessary to analyse the expression to ensure that certain conditions are fulfilled before the transformation can be applied. This can be illustrated by sum$'_{9B}$, where the first component of g's parameter will have the value 0 always, and therefore we may want to replace the true-branch of the conditional by Const[] 0[]. In the program so obtained it will then be possible to remove the first component of the parameter, since it will never be used. So for the sum$'_{9B}$ example we shall proceed in two stages

- apply a *forward* analysis, called *constant propagation*, to verify that the true-branch always will return 0—this will enable a program transformation that replaces it by Const[] 0[]; and then
- apply a *backward* analysis, called *liveness analysis*, to verify that the fixed point expression of the program only needs the second component of its parameter in order to compute the result—this will enable a program transformation that removes the first component of the parameter.

7.1 Constant propagation: the type part of $\mathscr{P}$

The purpose of *constant propagation* (Aho *et al.*, 1986) is to determine whether an expression will always evaluate to a constant, and to determine that constant. The analysis will be specified as an interpretation $\mathscr{P}$ following the pattern described in the previous section. So we shall define

- the meaning $\mathscr{P}_{t \underline{\rightarrow} t'}$ of the run-time types $t \underline{\rightarrow} t'$; and
- the meanings $\mathscr{P}_{\Box}$, $\mathscr{P}_{\texttt{Tuple}}$, etc., of the combinators and the fixed point.

The details of $\mathscr{P}$ will be very much as for $\mathscr{S}$, except that $\mathscr{P}$ will operate on *properties of values* rather than the 'real' values computed by $\mathscr{S}$. An expression of type $\underline{\texttt{Int}}$ will evaluate to an integer in $\mathscr{S}$, but in $\mathscr{P}$ it will evaluate to one of the properties

- $\mathbf{n}$, an integer, meaning that the 'real' value always will be equal to $\mathbf{n}$ (unless it is undefined);
- $\top$, meaning that we cannot determine a constant that the 'real' value always will be equal to, or
- $\bot$, meaning that the 'real' value always will be undefined.

We shall write $\mathbf{Z}^{\top}$ for this selection of properties, and a pictorial representation is given in fig. 4. We next define the *type part* of $\mathscr{P}$ by

$$\mathscr{P}_{t \underline{\rightarrow} t'} = [\![t]\!](\mathscr{P}) \rightarrow [\![t']\!](\mathscr{P})$$

so that the meaning of a run-time function will map properties of the argument type $[\![t]\!](\mathscr{P})$ to properties of the result type $[\![t']\!](\mathscr{P})$. The properties of values of type t are defined according to the structures of t, and we have

$$\begin{aligned}
[\![\underline{\texttt{Int}}]\!](\mathscr{P}) &= \mathbf{Z}^{\top} \\
[\![\underline{\texttt{Bool}}]\!](\mathscr{P}) &= \mathbf{T}^{\top} \\
[\![t \underline{\times} t']\!](\mathscr{P}) &= [\![t]\!](\mathscr{P}) \times [\![t']\!](\mathscr{P}) \\
[\![t\ \underline{\texttt{list}}]\!](\mathscr{P}) &= ([\![t]\!](\mathscr{P})^{*})^{\top}
\end{aligned}$$

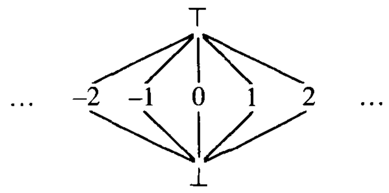

Fig. 4. Hasse diagram for $\mathbf{Z}^\top$.

Thus the properties of pairs of values are pairs of properties. The properties of lists of values will be lists of properties, but may also be the special value $\top$ that will be used as a property of lists with different lengths. With this definition it turns out that for all run-time types t the set $[\![t]\!](\mathscr{P})$ will contain an element (ambiguously denoted $\top$) representing that nothing can be determined about the 'real' value. It will also contain an element (ambiguously denoted $\bot$) representing that the 'real' value definitely will be undefined. In fact, each $[\![t]\!](\mathscr{P})$ will be a complete lattice whenever t is a run-time type. As in the standard semantics we have $[\![\mathtt{Sum}_1]\!](\mathscr{P}) = \mathbf{Z}^* \to \mathbf{Z}$, whereas $[\![\mathtt{Sum}_2]\!](\mathscr{P}) = ((\mathbf{Z}^\top)^*)^\top \to \mathbf{Z}^\top$, reflecting the difference between the annotations of $\mathtt{Sum}_1$ and $\mathtt{Sum}_2$.

7.2 *Constant propagation: the expression part of* $\mathscr{P}$

The *expression part* of $\mathscr{P}$ will specify how to operate on these properties. In fig. 5 we give a few illustrative clauses. The clauses for $\square$, `Tuple`, `Fst`, `Snd`, `Const` and `Id` are as in $\mathscr{S}$. The clauses for `Hd`, `Tl`, `Isnil` and $+$ are as in $\mathscr{S}$, but extended to cope with the special properties $\top$ and $\bot$. In the clause for the conditional we distinguish between whether the test evaluates to **true**, **false**, $\top$ or $\bot$. In the first two cases we choose the property of the appropriate branch. If the test evaluates to $\top$ then the 'real' value may be **true**, or it may be **false**, and we shall therefore combine the

$\mathscr{P}_\square\ (\mathbf{f},\mathbf{g})\ \mathbf{p} = \mathbf{f}(\mathbf{g}\ \mathbf{p})$

$\mathscr{P}_{\mathtt{Tuple}}\ (\mathbf{f},\mathbf{g})\ \mathbf{p} = \langle \mathbf{f}\ \mathbf{p}, \mathbf{g}\ \mathbf{p}\rangle$

$\mathscr{P}_{\mathtt{Fst}}\ \langle \mathbf{p},\mathbf{q}\rangle = \mathbf{p}$

$\mathscr{P}_{\mathtt{Snd}}\ \langle \mathbf{p},\mathbf{q}\rangle = \mathbf{q}$

$\mathscr{P}_{\mathtt{Hd}}\ \mathbf{p} = (\mathbf{p}=[\,])\to\bot \mid (\mathbf{p}=\mathbf{q}:\mathbf{p}')\to\mathbf{q} \mid (\mathbf{p}=\top)\to\top \mid \bot$

$\mathscr{P}_{\mathtt{Tl}}\ \mathbf{p} = (\mathbf{p}=[\,])\to\bot \mid (\mathbf{p}=\mathbf{q}:\mathbf{p}')\to\mathbf{p}' \mid (\mathbf{p}=\top)\to\top \mid \bot$

$\mathscr{P}_{\mathtt{Isnil}}\ \mathbf{p} = (\mathbf{p}=[\,])\to\mathbf{true} \mid (\mathbf{p}=\mathbf{q}:\mathbf{p}')\to\mathbf{false} \mid (\mathbf{p}=\top)\to\top \mid \bot$

$\mathscr{P}_{\mathtt{Cond}}\ (\mathbf{f},\mathbf{g},\mathbf{h})\ \mathbf{p} = (\mathbf{f}\ \mathbf{p}=\mathbf{true})\to\mathbf{g}\ \mathbf{p} \mid (\mathbf{f}\ \mathbf{p}=\mathbf{false})\to\mathbf{h}\ \mathbf{p} \mid$
$\qquad (\mathbf{f}\ \mathbf{p}=\top)\to(\mathbf{g}\ \mathbf{p}) \sqcup_\mathrm{I} (\mathbf{h}\ \mathbf{p}) \mid \bot$

$\mathscr{P}_{\mathtt{Const}}\ \mathbf{f}\ \mathbf{p} = \mathbf{f}$

$\mathscr{P}_{\mathtt{Id}}\ \mathbf{p} = \mathbf{p}$

$\mathscr{P}_{+}\ \langle \mathbf{p},\mathbf{q}\rangle = (\mathbf{p}=\bot \vee \mathbf{q}=\bot)\to\bot \mid (\mathbf{p}=\top \vee \mathbf{q}=\top)\to\top \mid \mathbf{p}+\mathbf{q}$

$\mathscr{P}^t_{\mathtt{fix}}\ \mathbf{f} = \mathbf{f}(\top) \quad$ for $t = t' \underline{\to} t''$

Fig. 5. Fragments of the expression part of $\mathscr{P}$.

properties obtained from the two branches using the least upper bound operation $\sqcup$. For $\mathbf{Z}^{\top}$, $\mathbf{p} \sqcup \mathbf{q}$ is defined to be $\top$ if $\mathbf{p}$ and $\mathbf{q}$ are distinct integers, whereas it is $\mathbf{p}$ if they are equal. Furthermore, $\mathbf{p} \sqcup \top = \top \sqcup \mathbf{p} = \top$ and $\mathbf{p} \sqcup \bot = \bot \sqcup \mathbf{p} = \mathbf{p}$ for all $\mathbf{p}$. Finally, the specification of $\mathscr{P}^{t}_{\mathtt{fix}}$ will determine how the fixed point is approximated. We shall take a rather crude approach, and assume that all recursive calls in the fixed point return non-constant values (i.e., $\top$).

If we apply $\mathscr{P}$ to `sum'`$_{9B}$ we then get

$$[\![\mathtt{sum'}_{9B}]\!](\mathscr{P})\mathbf{p} = (\mathbf{p} = [\,]) \to \mathbf{0} \mid (\mathbf{p} = \mathbf{q}:\mathbf{p'}) \to \top \mid (\mathbf{p} = \top) \to \top \mid \bot$$

7.3 The enabled program transformation

We cannot deduce very much from $[\![\mathtt{sum'}_{9B}]\!](\mathscr{P})$ about the values arising internally in `sum'`$_{9B}$. We shall therefore need a *sticky* variant of the analysis (Nielson, 1985, 1987; Hudak and Young, 1988) that will record the environments and the arguments supplied to $[\![e]\!](\mathscr{P})$ for the various subexpressions e of `sum'`$_{9B}$. For the true-branch `Fst[]` of the conditional we then get that $[\![\mathtt{Fst[]}]\!](\mathscr{P})$ is called with one environment and two different parameters, namely $\langle \mathbf{0},[\,]\rangle$ (if the test evaluates to **true**) and $\langle \mathbf{0},\top\rangle$ (if the test evaluates to $\top$). In both cases, $[\![\mathtt{Fst[]}]\!](\mathscr{P})$ will evaluate to $\mathbf{0}$, so we see that the analysis gives the expected result.

The constant propagation analysis enables a program transformation called *constant folding* (Aho *et al.*, 1986)

> *Assume* that e has a subexpression e' such that during the computation of $[\![e]\!](\mathscr{P})\mathit{env}$ all calls of $[\![e']\!](\mathscr{P})$ with the possible environments return the constant $\mathbf{v}$.
>
> *Then* e can safely be changed to the expression e'' obtained by replacing the subexpression e' by `Const[] v[]`.

Applying this transformation to `sum'`$_{9B}$ we get the program $\mathtt{sum}^{\mathrm{P}}_{2B}$

```
VAL fix(λg[ ].Cond(Isnil[ ]□Snd[ ], Const[ ] 0[ ],
                   +[ ]□Tuple(Hd[ ]□Snd[ ],
                              g□Tuple(Fst[ ],Tl[ ]□Snd[ ]))))
   □Tuple(Const[ ] 0[ ],Id[ ])
HAS Int list→Int
```

In a similar way, the remaining occurrence of `Fst[]` could be replaced by `Const[] 0[]`, but this is not necessary for the next transformation to take place. This transformation only preserves *partial correctness* in the sense that the original program may loop in situations where the transformed program will not loop.

7.4 Remarks

The interpretation $\mathscr{P}$ has been presented as a *non-standard* interpretation. However, in order to be sure that the transformation enabled by $\mathscr{P}$ is correct it is crucial to make sure that $\mathscr{P}$ only collects *safe* (i.e., correct) information. Because of decidability issues

$\mathscr{P}$ will, in general, not collect precise information, and the remedy then is to *err on the safe side*.

Safety can be formulated by specifying $\mathscr{P}$ as an *abstract interpretation*. There are several approaches to how to achieve this. One possibility is to define a concretization function mapping abstract values (i.e., those used by $\mathscr{P}$) to sets of concrete values (i.e., those used by $\mathscr{S}$). Another possibility is to use a representation function mapping concrete values to abstract values. The main use of these functions is to express safety which then must be proved to hold. In our case the proof will amount to verifying safety for all expressions in our notation (using structural induction), and thereby the result will follow for all programs.

We have developed this approach into a *meta-theory* that will ease the specification of the analyses and their safety proofs (Nielson, 1987, 1989). The idea is that *representation functions* are specified for the run-time base types, together with ways to pass between the run-time type constructors. This is sufficient to specify the type part of the interpretation and the representation functions for the various domains. The representation functions may be used to express safety, but they also provide means for constructing '*best induced analyses*', i.e., safe analyses that are as precise as possible among all safe interpretations with the same type part. As these analyses are not decidable, in general, the *meta-theory* also studies certain '*expected forms*' that may be used instead of the ones generated automatically and without compromising safety. It still remains to incorporate sticky analyses into this meta-theory.

To overcome the absence of total correctness in the program transformation enabled by constant propagation, one may investigate formulations of constant propagation based upon the approach given by Mycroft and Nielson (1983). This would allow us to let a property like **n** stand for the integer **n** *without* the need to allow undefinedness. The theory, however, becomes much more complex.

8 Backward abstract interpretation

In *liveness analysis* (Aho *et al.*, 1986) we want to know whether or not values are *live*, i.e., may be needed in future computations, or *dead*, i.e., definitely will not be needed in future computations. This analysis differs from the previous one in two important aspects. One is that we are not going to talk about properties of values but rather *properties of the future use of values*. Another difference is that the information about liveness propagates through the program in the *opposite direction* of the flow of control. Therefore, liveness analysis is often called a backward analysis, whereas constant propagation is called a forward analysis (Aho *et al.*, 1986).

8.1 Liveness analysis: type part of $\mathscr{L}$

The liveness analysis will be specified by an interpretation $\mathscr{L}$. A property of a value of type <u>`Int`</u> (or <u>`Bool`</u>) will either be

- **dead** if the value is definitely not needed in future computations; or
- **live** if the value may be needed later.

The *type part* of $\mathscr{L}$ will, e.g., have

$$[\![\underline{\texttt{int}}]\!](\mathscr{L}) = \{\textbf{dead}, \textbf{live}\}$$

$$[\![\underline{\texttt{Bool}}]\!](\mathscr{L}) = \{\textbf{dead}, \textbf{live}\}$$

$$[\![t \times t']\!](\mathscr{L}) = [\![t]\!](\mathscr{L}) \times [\![t']\!](\mathscr{L})$$

$$[\![t\,\underline{\texttt{list}}]\!](\mathscr{L}) = \{\textbf{dead}, \textbf{live}\}$$

So a property of the future use of a pair of values will be a pair of properties, one for each component. For the sake of simplicity, the properties of the future use of lists are defined to be {**dead, live**} so that it will not be possible to see if only parts of the list will be needed in future computations. If a more refined analysis is wanted we may, for example, replace the definition above with $[\![t\,\underline{\texttt{list}}]\!](\mathscr{L}) = \{\mathrm{L} \mid \mathrm{L} \subseteq [\![t]\!](\mathscr{L})^*\}$ (see, e.g., Nielson, 1989). With the definition above each set $[\![t]\!](\mathscr{L})$ will contain an element $\perp_t$ (ambiguously denoted **dead**) representing that the 'real' value will definitely not be needed in the future computations, and it will contain an element $\top_t$ (ambiguously denoted **live**) representing that the 'real' value may be needed in future computations. Because the analysis is a backward analysis, a property of functions will be a function that maps properties in the opposite direction

$$\mathscr{L}_{t \underline{\to} t'} = [\![t']\!](\mathscr{L}) \to [\![t]\!](\mathscr{L})$$

Then $[\![\texttt{Sum}_1]\!](\mathscr{L}) = \mathbf{Z}^* \to \mathbf{Z}$ and $[\![\texttt{Sum}_2]\!](\mathscr{L}) = \{\textbf{dead}, \textbf{live}\} \to \{\textbf{dead}, \textbf{live}\}$. The analysis is only specified for the subset of our notation where only first-order run-time functions are allowed. In fact, this was one of the motivations for the second transformation in subsection 4.3.

8.2 Liveness analysis: the expression part of $\mathscr{L}$

Turning to the *expression part* of $\mathscr{L}$ we shall specify how to operate on the properties above. In fig. 6 we give a few illustrative cases. The clause for $\Box$ reflects that functions have to be composed in the opposite order of the standard one. In the clause for

$$\mathscr{L}_{\Box}\ \mathbf{(f,g)\ p = g(f\ p)}$$
$$\mathscr{L}_{\texttt{Tuple}}\ \mathbf{(f,g)\ \langle p,q\rangle = (f\ p) \sqcup (g\ q)}$$
$$\mathscr{L}_{\texttt{Fst}}\ \mathbf{p = \langle p,dead\rangle}$$
$$\mathscr{L}_{\texttt{Snd}}\ \mathbf{p = \langle dead,p\rangle}$$
$$\mathscr{L}_{\texttt{Hd}}\ \mathbf{p = p}$$
$$\mathscr{L}_{\texttt{Tl}}\ \mathbf{p = p}$$
$$\mathscr{L}_{\texttt{Isnil}}\ \mathbf{p = p}$$
$$\mathscr{L}_{\texttt{Cond}}\ \mathbf{(f,g,h)\ p = (f\ p) \sqcup (g\ p) \sqcup (h\ p)}$$
$$\mathscr{L}_{\texttt{Const}}\ \mathbf{f\ p = dead}$$
$$\mathscr{L}_{\texttt{Id}}\ \mathbf{p = p}$$
$$\mathscr{L}_{+}\ \mathbf{p = \langle p,p\rangle}$$
$$\mathscr{L}^{t}_{\texttt{fix}}\ \mathbf{f = FIX\ (f)} \quad \text{for all } t$$

Fig. 6. Fragments of the expression part of $\mathscr{L}$.

`Tuple` we are given a pair ⟨**p,q**⟩ of properties for the future uses of the result of `Tuple(. . . , . . . )` and we use the least upper bound operation $\sqcup$ to join the results. On the set {**dead, live**} the operation is defined by **dead** $\sqcup$ **live** = **live** $\sqcup$ **dead** = **live** $\sqcup$ **live** = **live** and **dead** $\sqcup$ **dead** = **dead**. (This corresponds to the least upper bound in a partially ordered set where **dead** $\sqsubseteq$ **live**). The clauses for `Fst, Snd, Hd, Tl, Isnil, Const, Id` and + should be straightforward. For the conditional we simply combine the properties obtained from the test and the two branches. Finally, the meaning of the fixed point is defined to be the least fixed point as in the standard interpretation $\mathscr{S}$.

We can now apply $\mathscr{L}$ to the program $\mathtt{sum}^{\mathrm{P}}_{9\mathrm{B}}$ and get

$$[\![\mathtt{sum}^{\mathrm{P}}_{9\mathrm{B}}]\!](\mathscr{L})\,\mathbf{p} = \mathbf{p}$$

which simply states that if we need the sum of the elements of the list then we also need the list.

8.3 The enabled program transformation

As in the case of constant propagation, we shall need a sticky variant of this analysis. In the invocation of $[\![\mathtt{sum}^{\mathrm{P}}_{9\mathrm{B}}]\!](\mathscr{L})\,\mathbf{p}$ we then get that

$$[\![\mathtt{fix}\ (\lambda \mathrm{g}[\,]\ldots.)]\!](\mathscr{L})\,env$$

only is applied to the property **p** and always evaluates to ⟨**dead,p**⟩. This shows that the first component of the argument never is used.

Live variable analysis enables the following transformation on fixed points

Assume that e has a subexpression $\mathtt{fix}(\lambda g[t_1 \underline{\times}\, t_2 \underline{\rightarrow}\, t_3].e')$ such that during the computation of $[\![e]\!](\mathscr{L})\,env$ each call of $[\![\mathtt{fix}(\lambda g[t_1 \underline{\times}\, t_2 \underline{\rightarrow}\, t_3].e')]\!](\mathscr{L})$ with the possible environments return the property ⟨**dead,p**⟩ for some **p**.
Then e can safely be changed to e'' obtained by replacing the subexpression

$$\mathtt{fix}(\lambda g[t_1 \underline{\times}\, t_2 \underline{\rightarrow}\, t_3].e')$$

by

$$\mathtt{fix}(\lambda g'[t_2 \underline{\rightarrow}\, t_3].e'[g' \,\square\, \mathtt{Snd}[t_1 \underline{\times}\, t_2]/g] \,\square$$
$$\mathtt{Tuple}(\mathtt{Fix}[\,] \,\square\, (\mathtt{Curry\ Snd}[t_2 \underline{\times}\, t_1]), \mathtt{Id}[t_2])) \,\square\, \mathtt{Snd}[t_1 \underline{\times}\, t_2]$$

where g' is a fresh identifier.

Here $\mathtt{Fix}[\,] \,\square\, (\mathtt{Curry\ Snd}[t_2 \underline{\times}\, t_1])$ is a function that transforms the argument of type t_2 to an ('unneeded') value ($\perp$) of type t_1. Applying this transformation to $\mathtt{sum}^{\mathrm{P}}_{9\mathrm{B}}$ we get the program

```
VAL fix(λg'[ ].Cond(Isnil[ ] □Snd[ ], Const[ ] O[ ],
                  +[ ]□Tuple(Hd[ ]□Snd[ ],
                       g' □Snd[ ]□Tuple(Fst[ ], Tl[ ]□Snd[ ])))□
                  Tuple(Fix[ ]□(Curry Snd[ ]), Id[ ]))
      □Snd[ ]□Tuple(Const[ ] O[ ], Id[ ])
HAS Int list→Int
```

At this stage it is worth applying a few *algebraic transformations* (Section 5)

$$\texttt{Snd[]}\,\Box\,\texttt{Tuple}(e,e') \Rightarrow e'$$

$$e\,\Box\,\texttt{Id[]} \Rightarrow e$$

$$\texttt{Cond}(e_1,e_2,e_3)\,\Box\,e \Rightarrow \texttt{Cond}(e_1\,\Box\,e, e_2\,\Box\,e, e_3\,\Box\,e)$$

$$(\texttt{Const[]0[]})\,\Box\,e \Rightarrow \texttt{Const[]0[]}$$

$$\texttt{Tuple}(e_1,e_2)\,\Box\,e \Rightarrow \texttt{Tuple}(e_1\,\Box\,e, e_2\,\Box\,e)$$

and the resulting program is $\texttt{sum}^{\mathrm{L}}_{9\mathrm{B}}$

```
VAL fix(λg'[].Cond(Isnil[], Const[] 0[], +[]□Tuple(Hd[],
                                                    g'□Tl[])))
HAS Int list→Int
```

These transformations preserve the total correctness of the program.

8.4 Remarks

As in the previous section the analysis $\mathscr{L}$ is only specified as a non-standard interpretation, so nothing has been said about its safety properties. Unfortunately, only *some* of the techniques mentioned in Section 7 can be applied to $\mathscr{L}$. The main reason is that $\mathscr{L}$ is not concerned with properties of values but, as we mentioned earlier, properties of their future use. Thus it does not make sense to define representation functions as before. It is worth noticing that it is not the fact that $\mathscr{L}$ is a backward analysis that limits the applicability of the previous approach—indeed there are forward analyses (e.g., available expressions, Aho *et al.*, 1986) that cannot be proved safe with these techniques either. We refer to Neilson (1989) for further details about the correctness of $\mathscr{L}$.

It is also worth pointing out the relationship to the 'strictness continuations' used by Nielson and Nielson (1990*c*) and to the 'evaluators' of Burn (1987). There also appears to be a close relationship to the 'demand analysis' of Bjerner and Holmström (1989).

9 Code generation

One of the motivations behind the 2-level notation is that we should only generate code for run-time computations, not compile-time computations. We illustrate this by showing how to reinterpret the combinators so as to specify code generation for a simple abstract machine.

9.1 An abstract machine

The abstract machine we consider is closely related to the categorical abstract machine (Cousineau *et al.*, 1987), and the abstract machine of Nielson and Nielson (1986). The *configurations* of the machine has two components

- a *control stack* CS containing the code to be executed; and
- an *evaluation stack* ST containing the intermediate results.

$$
\begin{array}{l}
(\mathrm{ENTER}{:}\mathrm{CS},v{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},v{:}v{:}\mathrm{ST})\\
(\mathrm{SWITCH}{:}\mathrm{CS},v{:}u{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},u{:}v{:}\mathrm{ST})\\
(\mathrm{TUPLE}{:}\mathrm{CS},v{:}u{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},\langle v,u\rangle{:}\mathrm{ST})\\
(\mathrm{FST}{:}\mathrm{CS},\langle v,u\rangle{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},v{:}\mathrm{ST})\\
(\mathrm{SND}{:}\mathrm{CS},\langle v,u\rangle{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},u{:}\mathrm{ST})\\
(\mathrm{HD}{:}\mathrm{CS},[v{:}u]{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},v{:}\mathrm{ST})\\
(\mathrm{TL}{:}\mathrm{CS},[v{:}u]{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},u{:}\mathrm{ST})\\
(\mathrm{ISNIL}{:}\mathrm{CS},[\ ]{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},\mathbf{true}{:}\mathrm{ST})\\
(\mathrm{ISNIL}{:}\mathrm{CS},[v{:}u]{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},\mathbf{false}{:}\mathrm{ST})\\
(\mathrm{BRANCH}(\mathrm{C},\mathrm{C}'){:}\mathrm{CS},\mathbf{true}{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{C}{:}\mathrm{CS},\mathrm{ST})\\
(\mathrm{BRANCH}(\mathrm{C},\mathrm{C}'){:}\mathrm{CS},\mathbf{false}{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{C}'{:}\mathrm{CS},\mathrm{ST})\\
(\mathrm{CONST}(v){:}\mathrm{CS},u{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},v{:}\mathrm{ST})\\
(\mathrm{SKIP}{:}\mathrm{CS},\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},\mathrm{ST})\\
(\mathrm{ADD}{:}\mathrm{CS},\langle v,u\rangle{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},(v+u){:}\mathrm{ST})\\
(\mathrm{CALL}(l){:}\mathrm{CS},\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{ENV}(l){:}\mathrm{CS},\mathrm{ST})\\
(\mathrm{DELAY}(\mathrm{C}){:}\mathrm{CS},v{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},\{\mathrm{C},v\}{:}\mathrm{ST})\\
(\mathrm{RESUME}{:}\mathrm{CS},\{\mathrm{C},v\}{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{C}{:}\mathrm{CS},v{:}\mathrm{ST})\\
(\mathrm{RESUME}{:}\mathrm{CS},v{:}\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS},v{:}\mathrm{ST}) \quad \text{otherwise}
\end{array}
$$

Fig. 7. Abstract machine instructions.

In addition there is an *environment* ENV containing named pieces of code. During the execution of a program the configurations (CS,ST) will be modified, whereas the environment will remain unchanged. We write

$$(\mathrm{CS},\mathrm{ST}) \rightarrow_{\mathrm{ENV}} (\mathrm{CS}',\mathrm{ST}')$$

if one execution step of the machine will change the configuration from (CS,ST) to (CS',ST').

In fig. 7 we list some of the *instructions* of the machine, together with their operational meaning. The control stack is a list of instructions and we shall (ambiguously) write ':' for appending two lists and for prepending an element to a list. The evaluation stack is a list of values, and we distinguish between three different kinds of values

- primitive values such as the truth values and the integers;
- composite values such as pairs (denoted by $\langle \dots, \dots \rangle$) and lists (denoted by [...]); and
- delay closures (denoted by $\{\dots, \dots\}$) containing a list of instructions and a value.

A *delay closure* {C,**v**} is used to delay the execution of the code C on the stack with **v** on its top. The instruction DELAY(...) constructs a delay closure, whereas RESUME initiates the execution of the code of a delay closure on the given argument.

The overall operation of the machine upon input **v** and with program C is to start in the configuration (C,**v**: nil). It will then repeatedly execute the instructions of C and update the configuration accordingly. If at some state there is no next configuration then the machine stops. If one of the stacks of the configuration becomes empty or it calls a name not defined in the environment, then the machine also stops. It is, of course, also possible that the machine will continue executing instructions forever. However, if it stops and the configuration has the form (nil,**u**:nil) then we say that the machine has terminated with output **u**. In all other cases where the machine stops we shall say that it has terminated with failure.

As an example, consider the program ENTER:TUPLE:ADD. Assuming that the input is **7**, the initial configuration will be

$$(\text{ENTER}:\text{TUPLE}:\text{ADD}:\text{nil},\mathbf{7}:\text{nil})$$

Execution of the program with the empty environment $\varnothing$ will modify the configuration as follows

$$\begin{aligned}&(\text{ENTER}:\text{TUPLE}:\text{ADD}:\text{nil},\mathbf{7}:\text{nil})\\ &\quad\rightarrow_{\varnothing}(\text{TUPLE}:\text{ADD}:\text{nil},\mathbf{7}:\mathbf{7}:\text{nil})\\ &\quad\rightarrow_{\varnothing}(\text{ADD}:\text{nil},\langle\mathbf{7},\mathbf{7}\rangle:\text{nil})\\ &\quad\rightarrow_{\varnothing}(\text{nil},\mathbf{14}:\text{nil})\end{aligned}$$

The machine has now terminated with output **14**. The same effect is obtained by executing the program DELAY(ENTER:TUPLE:ADD):RESUME

$$\begin{aligned}&(\text{DELAY}(\text{ENTER}:\text{TUPLE}:\text{ADD}):\text{RESUME}:\text{nil},\mathbf{7}:\text{nil})\\ &\quad\rightarrow_{\varnothing}(\text{RESUME}:\text{nil},\{\text{ENTER}:\text{TUPLE}:\text{ADD}:\text{nil},\mathbf{7}\}:\text{nil})\\ &\quad\rightarrow_{\varnothing}(\text{ENTER}:\text{TUPLE}:\text{ADD}:\text{nil},\mathbf{7}:\text{nil})\\ &\quad\rightarrow_{\varnothing}(\text{TUPLE}:\text{ADD}:\text{nil},\mathbf{7}:\mathbf{7}:\text{nil})\\ &\quad\rightarrow_{\varnothing}(\text{ADD}:\text{nil},\langle\mathbf{7},\mathbf{7}\rangle:\text{nil})\\ &\quad\rightarrow_{\varnothing}(\text{nil},\mathbf{14}:\text{nil})\end{aligned}$$

9.2 Coding interpretation: the type part of $\mathscr{K}$

Recall that to specify the semantics it is sufficient to specify the meaning of the run-time function types, the combinators and the compile-time fixed point. Let **Code** be the set of instruction sequences for the stack machine. The *type part* of the interpretation $\mathscr{K}$ will then have

$$\mathscr{K}_{t \underline{\rightarrow} t'} = \mathbf{Code}$$

for all types $t \underline{\rightarrow} t'$, since we want to generate a sequence of instructions for any run-time function. Note that we do not use structural induction to define the meaning of the run-time types as in the previous sections. As an example, we have $[\![\texttt{Sum}_1]\!](\mathscr{K}) = \mathbf{Z}^* \rightarrow \mathbf{Z}$ and $[\![\texttt{Sum}_2]\!](\mathscr{K}) = \mathbf{Code}$. Perhaps more interestingly we have $[\![\texttt{Reduce}_9]\!](\mathscr{K}) = \mathbf{Code} \rightarrow \mathbf{Code}$, reflecting that given the code for the function argument of `reduce` we obtain the code for the specialized function. Intuitively, **Code** $\rightarrow$ **Code** is the type of code fragments with holes for where the argument code should be inserted.

9.3 Coding interpretation: the expression part of $\mathcal{K}$

The *expression part of $\mathcal{K}$* will specify how to generate code for the combinators and the fixed point. So, for example, $\mathcal{K}_{\mathtt{Fst}}$ will specify the code to be generated for `Fst[]` and $\mathcal{K}_{\square}$ will specify how to compose the two pieces of code for the arguments of $\square$.

The clauses of fig. 8 reflect that our language has a non-strict semantics in that the execution of the various operations are postponed (using the DELAY instruction) until they are really needed. We use a simple-minded approach where everything is enclosed in a delay closure whenever the result might possibly not be needed. As an example, the code for $e_1 \square e_2$ will contain the code for e_2, but encapsulated in a DELAY instruction because it is not known whether it needs to be executed. On the other hand, the code for e_1 needs to be executed in order to get a result, and is therefore not encapsulated. Similarly, we do not know whether the subexpressions e_1 and e_2 of `Tuple`(e_1, e_2) need to be evaluated, and therefore the corresponding fragments of code are enclosed in DELAY instructions.

Whenever we want to recover a value that may be a delay closure we use the instruction RESUME. As an example, the code for $\mathcal{K}_{\mathtt{Fst}}$ reflects that we must first evaluate the argument to a pair, and then we select the first component. The actual pair could very well be a pair of delay closures, and in that case the result will be further evaluated. The code for $\mathcal{K}_+$ shows how the RESUME instruction is used to force the evaluation of the argument to a pair, and then to force the evaluation of each of the components to an integer such that the addition instruction can be applied.

Finally, the code generated for the fixed point is simply CALL(f) where f is a labelled piece of code specified in the environment. The argument **f** of $\mathcal{K}^t_{\mathtt{fix}}$ is intuitively a code

$\mathcal{K}_{\square}$ **(f,g)** = DELAY **(g)**:**f**

$\mathcal{K}_{\mathtt{Tuple}}$ **(f,g)** = ENTER:DELAY **(g)**:SWITCH:DELAY **(f)**:TUPLE

$\mathcal{K}_{\mathtt{Fst}}$ = RESUME:FST:RESUME

$\mathcal{K}_{\mathtt{Snd}}$ = RESUME:SND:RESUME

$\mathcal{K}_{\mathtt{Hd}}$ = RESUME:HD:RESUME

$\mathcal{K}_{\mathtt{Tl}}$ = RESUME:TL:RESUME

$\mathcal{K}_{\mathtt{Isnil}}$ = RESUME:ISNIL

$\mathcal{K}_{\mathtt{Cond}}$ **(f,g,h)** = ENTER:**f**:BRANCH(**g,h**)

$\mathcal{K}_{\mathtt{Const}}$ **f** = CONST(**f**)

$\mathcal{K}_{\mathtt{Id}}$ = RESUME

$\mathcal{K}_+$ = RESUME:ENTER:SND:RESUME:
SWITCH:FST:RESUME:TUPLE:ADD

$\mathcal{K}^t_{\mathtt{fix}}$ **f** = CALL (f)

where t is $t' \underline{\rightarrow} t''$ and ENV is extended with
ENV (f) = **f**(CALL (f))
and where f is a fresh identifier

Fig. 8. Fragments of the expression part of $\mathcal{K}$.

fragment with *holes* in it for where the recursive calls have to be inserted, and in f(CALL(f)) the holes have been filled with CALL(f).

We can now apply $\mathscr{K}$ to the program $\text{sum}^{\text{L}}_{9\text{B}}$ and we obtain the code

```
CALL(sum)
```

The environment ENV will associate the label *sum* with the code

```
ENTER : RESUME : ISNIL :
BRANCH(CONST(0),
        DELAY(ENTER :
              DELAY(DELAY(RESUME : TL : RESUME) : CALL(sum)) :
              SWITCH :
              DELAY (RESUME : HD : RESUME) :
              TUPLE) :
        RESUME : ENTER : SND : RESUME : SWITCH : FST : RESUME : TUPLE : ADD)
```

9.4 Peephole optimization

The code generated by the coding interpretation $\mathscr{K}$ is rather inefficient, and we shall therefore show that a few optimization rules can be used to improve the quality of the code.

We shall use the following transformation rules, often called peep-hole optimizations, in order to reduce the number of delay closures

$$\text{DELAY(C)} : \text{RESUME} \Rightarrow \text{C}$$

$$\text{ENTER} : \text{C}_2 : \text{SWITCH} : \text{C}_1 : \text{TUPLE} : \text{ENTER} : \text{SND} : \text{C'}_2 : \text{SWITCH} : \text{FST} : \text{C'}_1 : \text{TUPLE}$$

$$\Rightarrow \text{ENTER} : \text{C}_2 : \text{C'}_2 : \text{SWITCH} : \text{C}_1 : \text{C'}_1 : \text{TUPLE}$$

Applying these transformations to the code of ENV(*sum*) we get

```
ENTER : RESUME : ISNIL :
BRANCH(CONST(0),
        ENTER : DELAY(RESUME : TL : RESUME) : CALL(sum) :
        SWITCH : RESUME : HD : RESUME : TUPLE : ADD)
```

Before each recursive call of the function the argument has been encapsulated in a delay closure. This is because the code generation reflects the non-strict semantics, and in the next section we shall see how a strictness analysis can be used to avoid generating these delay closures.

9.5 Remarks

The abstract machine has been chosen so as to make code generation straightforward. In our previous work (Nielson and Nielson, 1986, 1988*c*) we have used a more complicated machine where the code is linearized and the flow of control is modified by jumping between (uniquely generated) labels. However, the non-standard interpretation can also be used to generate code for full-fledged abstract machines like FAM (Cardelli, 1983) and the G-machine (Peyton Jones, 1987).

The semantics of the abstract machine is specified operationally, and when proving the correctness of the code generation we therefore have to relate a denotational semantics (specified by $\mathcal{S}$) to this operational semantics. Such a correctness proof is given by Nielson and Nielson (1988*c*) for the slightly more complicated machine mentioned above, and with a strict semantics (and thereby a code generation avoiding the generation of delay closures).

When the actions of the abstract machine are specified operationally it may be more convenient to prove the correctness of the code generation with respect to an *operational* semantics. This has been done by Cousineau *et al.* (1987) and Despeyroux (1986) for the categorical abstract machine, and it is interesting to note that the overall structure of that proof and the one in Nielson and Nielson (1988*c*) have much in common.

10 Optimization of the code generation

A closer inspection of the program `sum` shows that it is *strict* in its parameter. This is not reflected in the code generated in the previous section, so we shall want to improve the code generation. The idea is now

- to *specify* a strictness analysis; and
- to *modify* the coding interpretation to generate special code for strict functions.

A strictness analysis is just another *forward abstract interpretation*. Thus we shall proceed very much as in Section 7 when we studied the constant propagation analysis.

10.1 Strictness analysis

The purpose of strictness analysis (Mycroft, 1980, 1981; Burn *et al.*, 1986; Hughes, 1986; Nielson, 1987*b*; Wadler, 1987) is to determine whether an expression always will need its argument, i.e., whether it needs to be evaluated. The technical details of how $\mathcal{T}$ is specified are very much as those of $\mathcal{P}$ (and $\mathcal{S}$), except that the domains of properties are even simpler. An expression of type $\underline{\texttt{Int}}$ will now have one of two properties

- **1** meaning that the value may be defined or undefined (much as $\top$ in $[\![\underline{\texttt{Int}}]\!](\mathcal{P})$); or
- **0** meaning that the value is undefined (much as $\bot$ in $[\![\texttt{Int}]\!](\mathcal{P})$).

We shall write Δ for this set of properties. The *type part of* $\mathcal{T}$ will now define

$$\mathcal{T}_{t \underline{\rightarrow} t'} = [\![t]\!](\mathcal{T}) \rightarrow [\![t']\!](\mathcal{T})$$

so that the meaning of a run-time function will map properties of the argument type

$$\mathscr{T}_{\square}\ (\mathbf{f},\mathbf{g})\ \mathbf{p} = \mathbf{f}(\mathbf{g}\ \mathbf{p})$$
$$\mathscr{T}_{\texttt{Tuple}}\ (\mathbf{f},\mathbf{g})\ \mathbf{p} = \langle \mathbf{f}\ \mathbf{p}, \mathbf{g}\ \mathbf{p}\rangle$$
$$\mathscr{T}_{\texttt{Fst}}\ \langle \mathbf{p},\mathbf{q}\rangle = \mathbf{p}$$
$$\mathscr{T}_{\texttt{Snd}}\ \langle \mathbf{p},\mathbf{q}\rangle = \mathbf{q}$$
$$\mathscr{T}_{\texttt{Hd}}\ \mathbf{p} = \mathbf{p}$$
$$\mathscr{T}_{\texttt{Tl}}\ \mathbf{p} = \mathbf{p}$$
$$\mathscr{T}_{\texttt{Isnil}}\ \mathbf{p} = \mathbf{p}$$
$$\mathscr{T}_{\texttt{Cond}}\ (\mathbf{f},\mathbf{g},\mathbf{h})\ \mathbf{p} = (\mathbf{f}\ \mathbf{p}) \wedge ((\mathbf{g}\ \mathbf{p}) \vee (\mathbf{h}\ \mathbf{p}))$$
$$\mathscr{T}_{\texttt{Const}}\ \mathbf{f}\ \mathbf{p} = \mathbf{1}$$
$$\mathscr{T}_{\texttt{Id}}\ \mathbf{p} = \mathbf{p}$$
$$\mathscr{T}_{+}\ \langle \mathbf{p},\mathbf{q}\rangle = \mathbf{p} \wedge \mathbf{q}$$
$$\mathscr{T}^{t}_{\texttt{fix}}\ \mathbf{f} = \mathbf{f}(\mathbf{1}) \quad \text{for } t{=}t'\underline{\rightarrow}t''$$

Fig. 9. Fragments of the expression part of $\mathscr{T}$.

$[\![t]\!](\mathscr{T})$ to properties of the result type $[\![t']\!](\mathscr{T})$. The properties of values of type t are defined according to the structure of t

$$[\![\underline{\texttt{Int}}]\!](\mathscr{T}) = \Delta$$

$$[\![\underline{\texttt{Bool}}]\!](\mathscr{T}) = \Delta$$

$$[\![t\underline{\times}t']\!](\mathscr{T}) = [\![t]\!](\mathscr{T}) \times [\![t']\!](\mathscr{T})$$

$$[\![t\ \underline{\texttt{list}}]\!](\mathscr{T}) = \Delta$$

with the exception that the list types are interpreted as the simple domain Δ. This means that the analysis will not be sufficiently precise to determine whether some function is strict in, say, the head of the list or the spine of the list. This restriction is merely for the sake of simplicity.

The *expression part* of $\mathscr{T}$ will specify how to compute with these properties. In fig. 9 we give some of the more interesting cases. The operation $\mathbf{p} \vee \mathbf{q}$ is defined on Δ to be **1** if either **p** or **q** is **1** and otherwise **0**. In general, it is the least upper bound operation on the complete lattice $[\![t]\!](\mathscr{T})$ for t a run-time type. Similarly, $\mathbf{p} \wedge \mathbf{q}$ is defined on Δ to be **0** if either **p** or **q** is **0** and otherwise **1**. In general, it is the greatest lower bound operation. In the clause for $\mathscr{T}^{t}_{\texttt{fix}}$ we exploit that the complete lattice has a greatest element ⊤ (ambiguously denoted **1**).

If we apply $\mathscr{T}$ to $\texttt{sum}^{\texttt{L}}_{9\texttt{B}}$ we then get the expected

$$[\![\texttt{sum}^{\texttt{L}}_{9\texttt{B}}]\!](\mathscr{T})\,\mathbf{p} = \mathbf{p}$$

and taking **p** to be **0** this shows that $\texttt{sum}^{\texttt{L}}_{9\texttt{B}}$ is strict in its argument.

10.2 Modifying the coding interpretation

The idea is now to *combine* the strictness information and the code generation. We shall do that by defining an interpretation $\mathscr{C}$ that will compute the strictness

$\mathscr{C}_{\Box}$ ((f',f),(g',g)) = (λp . f'(g' p),
(f' 0 = 0)→ g:f | DELAY(g):f)

$\mathscr{C}_{\texttt{Tuple}}$ ((f',f)(g',g)) = (λp .⟨f' p, g' p⟩,
ENTER:DELAY(g):SWITCH:DELAY(f):TUPLE)

$\mathscr{C}_{\texttt{Fst}}$ = (λ⟨p,q⟩ . p, RESUME:FST:RESUME)

$\mathscr{C}_{\texttt{Snd}}$ = (λ⟨p,q⟩ . q, RESUME:SND:RESUME)

$\mathscr{C}_{\texttt{Hd}}$ = (λp,p, RESUME:HD:RESUME)

$\mathscr{C}_{\texttt{Tl}}$ = (λp,p, RESUME:TL:RESUME)

$\mathscr{C}_{\texttt{Isnil}}$ = (λp,p, RESUME:ISNIL)

$\mathscr{C}_{\texttt{Cond}}$ ((f',f), (g',g), (h',h)) = (λp . (f' p) ∧ ((g' p) ∨ (h' p)),
ENTER:f:BRANCH(g,h))

$\mathscr{C}_{\texttt{Const}}$ f = (λp . 1, CONST f)

$\mathscr{C}_{\texttt{Id}}$ = (λp . p,RESUME)

$\mathscr{C}_{+}$ = (λ⟨p,q⟩ . p ∧ q,
RESUME:ENTER:SND:RESUME:SWITCH:FST:RESUME:TUPLE:ADD)

$\mathscr{C}^{t}_{\texttt{fix}}$ F = let (f',f) = F(1,CALL(*f*))
in (f',CALL(*f*) where ENV(*f*) = f)
for $t = t' \underline{\rightarrow} t''$ and where *f* is a fresh identifier

Fig. 10. Fragments of the expression part of $\mathscr{C}$.

information at the same time as it computes the code to be generated. The *type part* of $\mathscr{C}$ will have

$$\mathscr{C}_{t \underline{\rightarrow} t'} = ([\![t]\!](\mathscr{T}) \rightarrow [\![t']\!](\mathscr{T})) \times \mathbf{Code}$$

where **Code** is as in the previous section. Thus the meaning of a run-time function will be a pair where the first component contains the strictness information and the second component is the code for the function.

The *expression part* of the interpretation will combine the clauses for the strictness analysis with those for the coding interpretation. In the clause specifying the code to be generated for $e_1 \Box e_2$ we use that if e_1 is strict, then e_2 *has to* be computed, and so we can dispense with the DELAY instruction. The clauses are shown in fig. 10.

We can now apply the coding interpretation to the program $\texttt{sum}^{\texttt{T}}_{9\texttt{B}}$ and we get the code

CALL(*sum*)

The environment ENV will now associate the label *sum* with the code

```
ENTER : RESUME : ISNIL :
BRANCH(CONST(0),
        ENTER : DELAY(RESUME : TL : RESUME : CALL(sum)) :
        SWITCH : DELAY(RESUME : HD : RESUME) :
        TUPLE :
        RESUME : ENTER : SND : RESUME : SWITCH : FST : RESUME : TUPLE : ADD)
```

We can apply the peephole optimizations of the previous section together with the transformations

$$\text{TUPLE}:\text{RESUME} \Rightarrow \text{TUPLE}$$

$$\text{ENTER}:\text{C}:\text{C}_1:\text{SWITCH}:\text{C}:\text{C}_2 \Rightarrow \text{C}:\text{ENTER}:\text{C}_1:\text{SWITCH}:\text{C}_2$$

and the result is

```
ENTER : RESUME : ISNIL :
BRANCH(CONST(0),
         RESUME : ENTER : TL : RESUME : CALL(sum) :
                  SWITCH : HD : RESUME : TUPLE : ADD)
```

10.3 Remarks

The code generation above can be further improved by introducing a test for *totality* (Mycroft, 1981). In the case of $e_1 \Box e_2$ we can then dispense with the DELAY instruction both in the case where e_2 is total and in the case where e_1 is strict. Furthermore, the DELAY instructions encapsulating the code generated for e_i for `Tuple`(e_1, e_2) can be avoided if e_i is total. However, such an improvement will not affect the code generated for our example program because the relevant functions will not be total. The formulation of the totality analysis is along the lines of the strictness analysis.

It has been shown how even better code can be generated (Nielson and Nielson, 1990*c*). One scheme amounts to using 'strictness continuations' and another scheme amounts to using 'evaluation degrees'.

We have taken the approach of *combining* two interpretations into one and then using results obtained by one part when defining the other part. An alternative approach would be to *annotate* the programs with strictness information and then extend the coding interpretation of Section 9 to cope with the annotated constructs.

11 Conclusion

To assess the applicability of our approach to the implementation of functional languages one will need to investigate the following aspects

- *efficiency*: the extent to which the code is comparable in efficiency (e.g., time or space) to that of code produced by other means;
- *correctness*: that the generated code behaves as expected, i.e., as the program being compiled; and
- *automation*: the extent to which the analyses and transformations can be performed without user interaction.

11.1 Efficiency

To obtain efficient code we have borrowed techniques for program analysis (or data flow analysis) and program transformation (or code optimization) from traditional compiler technology (Aho *et al.*, 1986). The functional programming style introduces an *implicit* distinction between binding times, as higher-order functions are typically supplied with some but not all of their arguments. This kind of distinction is often explicit in imperative languages, and the efficient implementations of these languages rely on this distinction. However, using a *binding time analysis* we can make this distinction *explicit* in a functional language as well, and then use it in the analysis and transformation of the programs. Following our approach the transformations at the source level may be grouped into four categories

- *Transformations aiding the binding time analysis*: the purpose is to get an appropriate distinction between binding times. A similar problem has been discussed in the context of semantic specifications (Nielson and Nielson, 1988*c*; Jørring and Scherlis, 1986).
- *Transformations at the compile-time level*: the purpose is to carry out some of the compile-time computations once and for all. These transformations are closely related to the partial evaluation described by Ershov (1982), and a restricted form of the transformations described in the unfold/fold framework of Burstall and Darlington (1977).
- *Transformations at the run-time level that are universally applicable*: the run-time level of our notation is converted into combinator notation, and we apply transformations similar to those developed for FP (Backus, 1978; Bellegarde, 1986; Harrison, 1988).
- *Transformations at the run-time level that are enabled by certain program analyses*: The enabling analyses are expressed as abstract interpretations (Nielson, 1987*a*, 1989), and they may express properties obtained by forward or backward analyses (Aho *et al.*, 1986). The use of transformations enabled by program analyses is discussed by Aho *et al.* (1986) and Nielson (1989).

The translation of the source program into target code is specified by a rather simple scheme, and to improve the efficiency of the code generated we need to analyse and transform the program

- *Analyse the program*: the purpose is here to collect information that can be used to select more appropriate code. The analyses may be specified as abstract interpretations (as in Section 10).
- *Transform the generated code*: we may use *peephole optimizations* to remove some of the superfluous instructions generated.

The example developed throughout this paper demonstrates that substantial improvements in the overall performance can be obtained using these techniques.

11.2 Correctness

Correctness involves

- validating that the (simple-minded) *code generation* scheme is correct;
- validating that the results of the *program analyses* are correct; and
- validating that the *program transformations* preserve the semantics of programs.

As we have seen, the analyses and transformations can be applied both at the source level and at the target level, i.e., one can perform analyses and transformations on the original functional program and on the code generated for it. As in traditional compiler technology, this means that the actual code generation is often kept rather simple-minded because the program may have been transformed so as to facilitate code generation, and any shortcomings in the generated code may be alleviated by subsequent transformations. This is important when one is arguing about the correctness issues, since it reduces the number of special cases to consider.

From the point of view of compiler construction, one is particularly interested in program transformations that are not semantics-preserving in general, but only in certain contexts. Information about the 'contexts' is provided by program analyses, and this motivates a close relationship between the program transformations and their enabling program analyses. An underlying principle of abstract interpretation is to approximate perfect information by 'erring on the safe side'. A number of abstract interpretations have been developed for functional languages (Burn *et al.*, 1986; Hughes, 1986, 1989; Nielson, 1987*b*) and a framework for developing abstract interpretations is described by Nielson (1987*a*, 1989) and Nielson and Nielson (1988*c*). The theoretical aspects of using information obtained from abstract interpretation to enable program transformations and to improve the code generation is discussed by Nielson (1985) and Hudak and Young (1988).

11.3 Automation

In an implementation of a programming language it is important that the user does not have to be concerned with the choice of analyses and transformations to be performed and the order in which to perform them. At the current stage our approach is certainly not amenable to automation. We have experimented with binding time analyses, combinator introduction, algebraic transformations and the concept of parameterized semantics in our *PSI-system*, and we expect that the system can be extended to include *specific* analyses and transformations. However, the range of analyses and transformations that one would want to perform is so broad that much more research is needed to determine which analyses and transformations should be built into an intelligent compiler for a functional language, and in which order they should be performed. We hope to look further into this using more general, but semi-automatic, systems such as those of Darlington and Pull (1988) and Feather (1982).

Acknowledgement

This work has been supported by the Danish Technical Research Council and the Danish Natural Science Research Council. A preliminary version of the earlier sections of this paper appeared in the 1989 Proceedings of the Functional Programming and Computer Architecture Conference.

References

Aho, A. V., Sethi, R. and Ullman, J. D. 1986. *Compilers – Principles, Techniques and Tools.* Addison-Wesley.

Backus, J. 1978. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Commun. ACM* **21**.

Bellegarde, B. 1986. Rewriting systems on FP expressions to reduce the number of sequences yielded. *Sci. Comput. Programming* **6**.

Bjerner, B. and Holmström, S. 1989. A compositional approach to time analysis of first order lazy functional programs. *Proc. Functional Programming Languages and Comput. Architectures.* ACM Press.

Burn, G. L., Hankin, C. and Abramsky, S. 1986. Strictness analysis for higher-order functions. *Sci. of Comput. Programming* **7**.

Burn, G. L. 1987. Evaluation transformers – a model for the parallel evaluation of functional languages. *Proc. Functional Programming Languages and Comput. Architectures Conf.* Volume 274 of *Lecture Notes in Computer Sciences.* Springer-Verlag.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *J. ACM* **24**.

Cardelli, C. 1983. *The functional abstract machine.* Bell Labs. Technical Report TR-107.

Cousineau, G., Curien, P.-L. and Mauny, M. 1987. The categorical abstract machine. *Sci. Comput. Programming* **8**.

Curien, P.-L. 1986. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman.

Damas, L. 1985. *Type assignment in programming languages.* PhD Thesis CST-33-85, University of Edinburgh, Scotland.

Darlington, J. and Pull, H. 1988. A program development methodology based on a unified approach to execution and transformation. In D. Bjørner *et al.* (editors), *Partial Evaluation and Mixed Computation.* North-Holland.

Despeyroux, J. 1986. Proof of translation in natural semantics. *Symposium on Logic in Computer Science.* IEEE Computer Society Press.

Ershov, A. P. 1982. Mixed computation: potential applications and problems for study. *Theor. Comput. Sci.* **18**.

Feather, M. S. 1982. A system for assisting program transformation. *ACM Trans. Programming Languages and Systems* **4**.

Harrison, P. G. 1988. Linearisation: an optimisation for nonlinear functional programs. *Sci. of Comput. Programming* **10**.

Hughes, J. 1982. Supercombinators: a new implementation method for applicative languages. *Proc. 1982 ACM Conf. on LISP and Functional Programming.* ACM Press.

Hughes, J. 1986. Strictness detection in non-flat domains. *Proc. Programs as Data Objects*, Volume 217 of *Lecture Notes in Computer Science.* Springer-Verlag.

Hughes, J. 1988. Backwards analysis of functional programs. In D. Bjørner *et al.* (editors), *Partial Evaluation and Mixed Computation.* North-Holland.

Hudak, P. and Young, J. 1988. A collecting interpretation of expressions (without powerdomains). *Proc. 15th ACM Symposium on Principles of Programming Languages* ACM Press.

Jones, N. D., Sestoft, P. and Søndergaard, H. 1985. An experiment in partial evaluation: the generation of a compiler generator. *Proc. Rewriting Techniques and Applications.* Volume 202 of *Lecture Notes in Computer Science.* Springer-Verlag.

Jørring, U. and Scherlis, W. L. 1986. Compilers and staging transformations. *Proc. 13th ACM Symposium on Principles of Programming Languages.* ACM Press.

Lambek, J. and Scott, P. J. 1986. *Introduction to Higher Order Categorical Logic.* Cambridge Studies in Advanced Mathematics 7. Cambridge University Press.

Milner, R. 1984. The standard ML core language. *Proc. ACM Conference on LISP and Functional Programming*. ACM Press.

Milner, R. 1978. A theory of type polymorphism in programming. *J. Comput. Systems* **17**.

Mogensen, T. 1989. Binding time analysis for higher order polymorphically typed languages. *TAPSOFT 1989, Lecture Notes in Computer Science*. Springer-Verlag.

Montenyohl, M. and Wand, M. 1988. Correct flow analysis in continuation semantics. *Proc. 15th ACM Symposium on Principles of Programming Languages*. ACM Press.

Mycroft, A. 1981. *Abstract interpretation and optimizing transformations for applicative programs*. PhD Thesis CTS-15-81, University of Edinburgh, Scotland.

Mycroft, A. 1980. The theory and practice of transforming call-by-need into call-by-value. *Proc. 4th International Symposium on Programming*. Volume 83 of *Lectures Notes in Computer Science*. Springer-Verlag.

Mycroft, A. and Nielson, F. 1983. Strong abstract interpretation using powerdomains. *Proc. ICALP 1983*. Volume 154 of *Lecture Notes in Computer Science*. Springer-Verlag.

Nielson, F. 1985. Program transformations in a denotational setting. *ACM Trans. Programming Languages and Systems* **7**.

Nielson, H. R. and Nielson, F. 1986. Semantics directed compiling for functional languages. *Proc. ACM Conf. on LISP and Functional Programming*. ACM Press.

Nielson, F. 1987*a*. Towards a denotational theory of abstract interpretation. In S. Abramsky and C. Hankin (editors), *Abstract Interpretation of Declarative Languages*. Ellis Horwood.

Nielson, F. 1987*b*. Strictness analysis and denotational abstract interpretation (extended abstract). *ACM Conf. on Principles of Programming Languages*. ACM Press.

Nielson, F. 1988. A formal type system for comparing partial evaluators. *Partial Evaluation and Mixed Computation*. North-Holland.

Nielson, H. R. and Nielson, F. 1988*a*. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming* **10**. (Also see *Proc. 15th ACM Symposium on Principles of Programming Languages*. ACM Press.)

Nielson, F. and Nielson, H. R. 1988*b*. 2-level λ-lifting. *Proc. ESOP 1988*. Volume 300 of *Lecture Notes in Computer Science*. Springer-Verlag.

Nielson, F. and Nielson, H. R. 1988*c*. Two-level semantics and code generation. *Theor. Comput. Sci.* **56**.

Nielson, F. 1989. Two-level semantics and abstract interpretation. *Theor. Comput. Sci.* **69**.

Nielson, H. R. and Nielson, F. 1989. Transformations on higher-order functions. *Proc. Functional Programming and Computer Architecture Conference*. ACM Press.

Nielson, H. R. and Nielson, F. 1990*a*. Functional completeness of the mixed λ-calculus and combinatory logic. *Theor. Comput. Sci.* **70**.

Nielson, H. R. and Nielson, F. 1990*b*. Eureka definitions for free! *or* Disagreement points for fold/unfold transformations. *Proc. ESOP '90*. Volume 432 of *Lecture Notes in Computer Science*. Springer-Verlag.

Nielson, H. R. and Nielson, F. 1990*c*. Context information for lazy code generation. *Proc. LISP and Functional Programming Conference*. ACM Press.

Peyton Jones, S. 1987. *The Implementation of Functional Programming Languages*. Prentice Hall.

Schmidt, D. A. 1988. Static properties of partial evaluation. *Partial Evaluation and Mixed Computation*. North-Holland.

Turner, D. 1979. A new implementation technique for applicative languages. *Software, Practice and Experience* **9**.

Turner, D. A. 1985. Miranda: a non-strict functional language with polymorphic types. *Proc. Functional Programming Languages and Computer Architectures*. Volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag.

Wadler, P. 1987. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C. Hankin (editors), *Abstract Interpretation of Declarative Languages*. Ellis Horwood.