Transformations on higher-order functions

Hanne Riis Nielson*
Flemming Nielson†

Abstract

Traditional functional languages do not have an explicit distinction between binding times. It arises implicitly, however, as typically one instantiates a higher-order function with the known arguments whereas the unknown arguments still are to be taken as parameters. The distinction between 'known' and 'unknown' is closely related to the distinction between binding times, e.g. the distinction between compile-time and run-time. We shall therefore use a combination of polymorphic type inference and binding time analysis to obtain the required information.

Following the current trend in the implementation of functional languages we shall then transform the run-time level (not the compile-time level) of the program 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.

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 one of its instances by verifying that the arguments satisfy some simple properties.

One of the disadvantages is that although a compiler may reuse code generated for higher-order functions the resulting efficiency is often rather poor. The reason is that when generating code for the higher-order function it is impossible to make any assumptions about its arguments and to optimise the code accordingly. Also conventional machine architectures makes it rather costly to use functions as arguments.

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 the implementation of functional languages

- annotate the programs with type information so that they can be uniquely typed (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), and
- specify various abstract interpretations by reinterpreting the combinators and use these analyses to enable program transformations that are not applicable under all circumstances (Section 6, 7 and 8).

We shall illustrate our approach for an example program in the enriched $\lambda$-calculus presented in Section 2. Finally, Section 9 contains the concluding remarks.

2 Operations made explicit

In MIRANDA [29] the functions reduce and sum may be written as

\[
\text{reduce } f \ u = g \\
\text{where } g \ [] = u \\
g \ (x:xs) = f \ x \ (g \ xs)
\]

\[
\text{sum} = \text{reduce} \ (+) \ 0
\]

The left hand side of an equation specifies the name of the function and a list of patterns for its parameters. The right hand side is the body of the function. If more than one equation is given for the same function (as for $g$) then there is an implicit conditional testing the form of the patterns in the parameter list. Also recursion is left implicit as the occurrence of a function name on the left hand side as well as the right hand side of an equation indicates that the corresponding function is recursively defined (as happens for $g$). Finally, function application is left implicit as the function is just juxtaposed with its arguments.

A similar equational definition of functions is allowed in STANDARD ML [13]. However here one has the possibility of making some of the implicit operations or concepts more explicit as is illustrated in

\[
\text{val reduce = fn } f \Rightarrow fn \ u = \Rightarrow
\]

\[
\text{let val rec } g = fn \ xs = \Rightarrow
\]

\[
\text{if } xs = [] \text{ then } u \Rightarrow
\]

\[
\text{else } f \ (\text{hd } xs) \ (g \ (\text{tl } xs)) \Rightarrow
\]

\[
\text{in } g \ \text{end};
\]

\[
\text{val sum} = \text{reduce} \ (fn \ x \Rightarrow fn \ y \Rightarrow x+y) \ 0;
\]

Function abstraction is now expressed explicitly by the construct \( \text{fn } \ldots \Rightarrow \ldots \) and the recursive structure of $g$ is expressed by the explicit occurrence of $\text{rec}$. Also the test on the form of the list argument is expressed explicitly.

Our approach requires all the operations to be expressed explicitly and so we will need to be based on a notation where programs are closer to the STANDARD ML program displayed above than the MIRANDA program. We shall therefore define a small language, an enriched $\lambda$-calculus, that captures a few of the more important constructs present in modern functional languages like MIRANDA and STANDARD ML and does so in an explicit way. The formal development is then to be performed for that language and in many cases it will be straight-forward to extend it to larger languages.

In the enriched $\lambda$-calculus we shall write the program sum as

\[
\text{DEF } \text{reduce} = \\
\lambda f.\lambda u.\text{fix} \ (\lambda g.\lambda xs.
\]

\[
\text{if isnil } xs \text{ then } u \\
\text{else } f \ (\text{hd } xs) \ (g \ (\text{tl } xs))
\]

\[
\text{VAL } \text{reduce} \ (\lambda x.\lambda y.\ldots(x,y)) \ (0)
\]

\[
\text{HAS } \text{Int list } \rightarrow \text{ Int}
\]
Here we use parentheses for function application, \( \text{fix} \) to make recursive definitions and angle brackets to construct pairs.

3 Types made explicit

Both Miranda and Standard ML have 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 \( \text{reduce} \) is

\[
(\alpha \to \beta \to \beta) \to \alpha \text{ list} \to \beta
\]

where \( \alpha \) and \( \beta \) are so-called type variables. The occurrence of \( \text{reduce} \) in the definition of \( \text{sum} \) has the type

\[
(\text{Int} \to \text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \text{ list} \to \text{Int}
\]

because it is applied to arguments of type \( \text{Int} \to \text{Int} \to \text{Int} \) and \( \text{Int} \).

The enriched \( \lambda \)-calculus is equipped with a type inference system closely related to that found in Miranda and Standard ML [14,6]. The type inference is based upon a few rules for how to build well-formed expressions. As an example, the function application \( e(e') \) is only well-formed if the type of \( e \) has the form \( t \to t' \) and if the type of \( e' \) is \( t \) and then the type of the application is \( t' \). Similar rules exist for the other composite constructs of the language. For the constants we have axioms stating e.g. that \( + \) has type \( \text{Int} \times \text{Int} \to \text{Int} \) and that \( 0 \) has type \( \text{Int} \). Based upon such axioms and rules we can infer that the program \( \text{sum} \) is well-formed and we can determine the types of the various subexpressions. Of course the results obtained are the same as those mentioned above for Miranda and Standard ML.

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 \( \lambda \)-abstractions. This means that type analysis will transform the untyped program for \( \text{sum} \) (of Section 2) into the following typed program:

\[
\text{DEF reduce} = \lambda f[\text{Int} \to \text{Int} \to \text{Int}], \lambda u[\text{Int}].\text{fix} (\lambda g[\text{Int list} \to \text{Int}], \lambda x[\text{Int list}] . \\
\text{if isnil x} \text{ then } u \\
\text{else } f(\text{hd x})(g(\text{tl x}))
\]

\[
\text{VAL reduce} = (\lambda x[\text{Int}], \lambda y[\text{Int}] . + [\text{Int} \times \text{Int} \to \text{Int}](x,y)) \\
(0[\text{Int}])
\]

\[
\text{HAS Int list} \to \text{Int}
\]

Note that the type of \( \text{reduce} \) now is fixed. This is possible because there is only one application of \( \text{reduce} \) in the program. Otherwise we may have to duplicate the definition of some of the functions as the enriched \( \lambda \)-calculus does not (yet) allow the use of type variables.

4 Binding time made explicit

Neither Miranda nor Standard ML nor the enriched \( \lambda \)-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 \( \lambda \)-calculus accordingly.

4.1 2-level syntax

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

\[
\text{Sum}_t = \text{Int list} \to \text{Int}
\]

The fact that the list argument is not available at compile-time will have consequences for when the parameters are available for \( \text{reduce} \). Again we shall record this by underlining parts of the type

\[
\text{Reduce}_t = (\text{Int} \to \text{Int} \to \text{Int}) \to \text{Int} \to \text{Int list} \to \text{Int}
\]

\( \text{Sum}_t \) and \( \text{Reduce}_t \) are examples of 2-level types.

We shall 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 it is clear that Sumt and Reducet are unacceptable annotations because they denote functions that get one of their arguments at run-time but nonetheless are able to deliver the results at compile-time. Our point of view shall therefore be that Sumt and Reducet 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 Int list whereas we can manipulate objects of type Int list → Int because the latter type may be regarded as the type of code for functions (to be executed at run-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

$$\text{Sum}_1 = \text{Int list} \rightarrow \text{Int}$$
$$\text{Sum}_2 = \text{Int list} \Rightarrow \text{Int}$$

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

<table>
<thead>
<tr>
<th>Reduce</th>
<th>Underlining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce</td>
<td>(Int → Int → Int) → Int → Int list → Int</td>
</tr>
<tr>
<td>Reduce</td>
<td>(Int → Int → Int) → Int → Int list → Int</td>
</tr>
<tr>
<td>Reduce</td>
<td>(Int → Int → Int) → Int → Int list → Int</td>
</tr>
<tr>
<td>Reduce</td>
<td>(Int → Int → Int) → Int → Int list → Int</td>
</tr>
<tr>
<td>Reduce</td>
<td>(Int → Int → Int) → Int → Int list → Int</td>
</tr>
<tr>
<td>Reduce</td>
<td>(Int → Int → Int) → Int → Int list → Int</td>
</tr>
<tr>
<td>Reduce</td>
<td>(Int → Int → Int) → Int → Int list → Int</td>
</tr>
<tr>
<td>Reduce</td>
<td>(Int → Int → Int) → Int → Int list → Int</td>
</tr>
</tbody>
</table>

Table 1: Well-formed 2-level types for reduce

Figure 1: Compatible 2-level types for reduce

We shall say that two 2-level types have the same underlying type if they are equal 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 also are 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 Reduce3 is compatible with Reduce1 but Reduce2 is not. This is expressed by the Hasse diagram of Figure 1.

We shall say that the best completion of Reduce1
is Reduce\textsubscript{3} because all other well-formed 2-level types that are compatible with Reduce\textsubscript{0} also will be compatible with Reduce\textsubscript{3}. (Shortly we shall see that the actual definition of reduce will impose further restrictions upon the type.) Similarly, the best completion of Sum\textsubscript{0} is Sum\textsubscript{3}. A formal account may be found in [22].

4.2 Binding time analysis

Given the binding time information expressed by a 2-level type as e.g. Sum\textsubscript{0} we shall now annotate the complete program so that we can see 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 discussion above suggests that the annotation of reduce should reflect the binding time information given by Reduce\textsubscript{3}. However, it is not possible to annotate the definition of reduce so that it is both well-formed and will have type Reduce\textsubscript{3}. The reason is that the binding times do not match: according to Reduce\textsubscript{3} the function argument \( f \) has type \( \text{Int} \to \text{Int} \to \text{Int} \) and the list argument \( xs \) has type \( \text{Int list} \) so \( f(hd \ xs) \) will supply \( f \) with a run-time argument although it expects a compile-time argument. This will be formalized by defining a well-formedness condition on 2-level expressions [23]. There are two ingredients in this. One is that the underlying expression (obtained by removing the underlinings) must type-check properly. The other ingredient is that the binding times must agree. For our example this means that if the first argument of \( f \) only is available at run-time (i.e. has type \( \text{Int} \)) then also the 2-level type of \( f \) must have the first occurrence of \( \text{Int} \) underlined.

Given an incomplete annotation \( \text{sum} \) of sum

\[
\text{DEF reduce} = \\
\lambda f[\text{Int} \to \text{Int} \to \text{Int}], \lambda u[\text{Int}]. \\
\text{fix} (\lambda g[\text{Int list} \to \text{Int}]. \lambda x[\text{Int list}]. \\
\text{if isnil} \ xs \ \text{then} \ u \\
\text{else} \ f(hd \ xs)(g(tl \ xs))) \\
\text{VAL reduce} \\
(\lambda x[\text{Int}]. \lambda y[\text{Int}]. [+\text{Int} \times \text{Int} \to \text{Int}](x,y)))
\]

the binding time analysis [21,22,27] 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 \( \text{sum} \) is \( \text{sum} \)

\[
\text{DEF reduce} = \\
\lambda f[\text{Int} \to \text{Int} \to \text{Int}], \lambda u[\text{Int}]. \\
\text{fix} (\lambda g[\text{Int list} \to \text{Int}]. \lambda x[\text{Int list}]. \\
\text{if isnil} \ xs \ \text{then} \ u \\
\text{else} \ f(hd \ xs)(g(tl \ xs))) \\
\text{VAL reduce} \\
(\lambda x[\text{Int}]. \lambda y[\text{Int}]. [+\text{Int} \times \text{Int} \to \text{Int}](x,y))) \\
(\lambda [0[\text{Int}])
\]

\#HAS \text{Int list} \to \text{Int}\

where \( \text{reduce} \) has type \( \text{Reduce} \). Here we cannot use e.g. \( \lambda u[\text{Int}] \) instead of \( \lambda u[\text{Int}] \) as the well-formedness conditions [23] on types only allow us to manipulate run-time function types at compile-time and not run-time data types (like \( \text{Int} \)).

The remaining well-formed 2-level programs are less interesting. Corresponding to the 2-level type \( \text{Reduce} \) we have a program with no underlinings at all and corresponding to \( \text{Reduce} \) we have a program where all operations are underlined. These are the only well-formed 2-level programs with sum as the underlying program\(^3\). Note that in each of these programs the binding times for 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 of the function.

\(^3\) We apply an adapted version of the binding time analysis algorithm of [22] in order to comply with the well-formedness predicate of [23].

\(^4\) This statement holds for the well-formedness predicate of [23]. If the predicate of [22,21] is used then there will be two more well-formed programs.
4.3 Transformations aiding the binding time analysis

In \texttt{sum}_9 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 data flow analyses and when generating code. We may therefore want to replace the \textit{run-time fixed point} by a \textit{compile-time fixed point}. Since \texttt{sum}_9 is the best completion of \texttt{sum}_8 we cannot obtain this effect by simply changing the annotation. The idea is therefore first

- to \texttt{transform} the underlying program, and then

- to \texttt{repeat} the binding time analysis.

In our case we shall apply the transformation

\[ \lambda z[t_1]. \text{fix} (\lambda f[t_2 \to t_3]. e) \Rightarrow \text{fix} (\lambda g[t_1 \to t_2 \to t_3]. \lambda z[t_1]. e[g(z)/f]) \]

that replaces the first pattern with the second. Here \(e[g(z)/f]\) is \(e\) with all occurrences of \(f\) replaced by \(g(z)\) and \(g\) is assumed to be ‘fresh’. We shall then repeat the binding time analysis with \texttt{Sum}_b as the ‘goal’ type and get

\[ \text{DEF} \ \text{reduce}_{gb} = \lambda f[\text{Int} \to \text{Int} \to \text{Int}]. \]

\[ \lambda u[\text{Int}]. \lambda x[\text{Int list}]. \]

\[ \text{fix} (\lambda g[\text{Int} \times \text{Int list} \to \text{Int}]. \lambda z[\text{Int} \times \text{Int list}]. \]

\[ \text{if isnil (snd z) then \text{fst} z \}
\]

\[ \text{else f (hd (snd z))}
\]

\[ (g ((\text{fst} z, \text{tl (snd z)}))((x, y))) \]

\[ ((u, x)) \]

\[ \text{VAL} \ \text{reduce}_{gb} = \lambda x[\text{Int}]. \lambda y[\text{Int}]. +[\text{Int} \times \text{Int} \to \text{Int}]((x, y)) \]

\[ (0[\text{Int}]) \]

\[ \text{HAS Int list \to Int} \]

so we see that the fixed point now is computed at compile-time and that the function \(g'\) expressing the recursive structure of the body 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. It is well-known that, at least for stack-based implementations (see e.g. [9]), it is expensive to handle higher-order functions so we may want to transform the program further to change the functionality of \(g'\).

To do that we shall first apply the following transformation to the underlying program of \texttt{sum}_9

\[ \text{fix} (\lambda f[t_1 \to t_2 \to t_3]. \lambda x[t_1]. \lambda y[t_2]. e) \Rightarrow \lambda x[t_1]. \lambda y[t_2]. (\text{fix} (\lambda g[t_1 \times t_2 \to t_3]. \lambda x[t_1 \times t_2].
\]

\[ e[\lambda z[t_1]. \lambda y[t_2]. g((z, y))/f]
\]

\[ [\text{fst} z/z][\text{snd} z/y]) \]

\[(x, y)) \]

where \(g\) and \(z\) are assumed to be ‘fresh’. Next we shall apply the \(\beta\)-transformation

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

The binding time analysis is then applied to the resulting program and \texttt{Sum}_b as the overall annotated type and we get the program \texttt{sum}_{9b}

\[ \text{DEF} \ \text{reduce}_{gb} = \lambda f[\text{Int} \to \text{Int} \to \text{Int}]. \]

\[ \lambda x[\text{Int}]. \lambda y[\text{Int}]. +[\text{Int} \times \text{Int} \to \text{Int}]((x, y)) \]

\[ (0[\text{Int}]) \]

\[ \text{HAS Int list \to Int} \]

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 straight-forward to do so when the run-time computations are expressed in the form of \(\lambda\)-expressions. As an example, the usual meaning of
\[ \lambda x[\text{Int} \times \text{Int}], f (g \langle x \rangle) \]

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 [2,5]) rather than values and functions. Then we would write

\[ f \circ g \]

for the expression above and the effect of both \(\lambda v.f\ (g\ v)\) and \(\lambda v.g\ (f\ v)\) can be obtained by reinterpreting the functional \(\square\). This observation then calls for transforming the run-time computations into combinator form. This is analogous to the motivation behind the use of combinators in the implementation of functional languages [28]. It is also analogous to the use of categorical combinators when the typed \(\lambda\)-calculus is given a semantics in an arbitrary cartesian closed category [12]. However, what distinguishes our motivation from these analogous motivations is that we shall leave the compile-time computations in the form of \(\lambda\)-expressions and only transform the run-time computations into combinator form.

5.1 Combinator introduction

We shall say that 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 \(\text{sum}_{gB}\) the corresponding program \(\text{sum}_{\text{GB}}\) in combinator form will be

\[
\text{DEF } \text{reduce}_{gB} = \lambda f[\ ], \text{Curry} (\text{fix}(\lambda g[\ ]),
\text{Cond}(\text{Isnil}[\ ] \circ \text{Snd}[\ ], \text{Fst}[\ ],
\text{Apply}[\ ] \circ \text{Tuple}(f \circ \text{Rd}[\ ] \circ \text{Snd}[\ ],
g \circ \text{Tuple}(\text{Fst}[\ ], \text{Tl}[\ ] \circ \text{Snd}[\ ]))))
\]

\[
\text{VAL } \text{Apply}[\ ] \circ
\text{Tuple}(\text{reduce}_{gB} (\text{Curry} \ast[\ ])) \circ
(\text{Const}[\ ] \circ [\ ], \text{Id}[\ ])
\]

5.2 Partial evaluation and algebraic transformations

Before turning to the interpretation of the combinators it may be worthwhile to observe that the program \(\text{sum}_{gB}\) can be further simplified. At the
compile-time level we can supply the occurrence of reduce with its first parameter (which is known at compile-time). This sort of program transformation is often called partial evaluation [7,21] and in our case it gives rise to the program

$$\text{VAL Apply}\[\text{ ] } \square \text{Tuple((Curry (fix (\lambda g[ ).
\text{Cond}(\text{Isnil[ ] } \square \text{Snd[ ], Fst[ ]},
\text{Apply[ ] } \square \text{Tuple((Curry +[ ] )} \text{HD[ ]} \square \text{Snd[ ]},
\text{g } \square \text{Tuple(Fst[ ], TL[ ] } \square \text{Snd[ ])}))))\)
\square (\text{Const[ ] 0[ ]}, \text{Id[ ]}) $$

\text{HAS Int list } \rightarrow \text{Int}

The run-time counterpart of partial evaluation is called algebraic transformations [2]. An example is the transformation

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

If we apply this transformation twice to the program above then we get the program $\sum_{9B}$

$$\text{VAL fix}(\lambda g[ ).\text{Cond}(\text{Isnil[ ] } \square \text{Snd[ ], Fst[ ]},
\text{+}[ ] \text{Tuple(Hd[ ] } \square \text{Snd[ ]},
\text{g } \text{Tuple(Fst[ ], TL[ ] } \square \text{Snd[ ]))})
\square \text{Tuple(\text{Const[ ] 0[ ]}, \text{Id[ ]}) $$

\text{HAS Int list } \rightarrow \text{Int}

Note that by now all higher-order run-time functions have disappeared.

6 Parameterized semantics

Recall that 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 \[17,19,20,25\] 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).

Then there will be a standard way of extending the interpretation to a semantics defining the meaning of all well-formed 2-level types, and the meaning of all well-formed 2-level programs (and 2-level expressions) in combinator form. In the next sections we shall give some example interpretations for specifying forward and backward data flow analysis and in \[17,20\] it is shown how to specify code generation. The correctness of these interpretations will be relative to a non-strict (or lazy) standard interpretation that will be given shortly.

6.1 The type part of an interpretation

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

\[S_{\rightarrow t'}(S) = [t](S) \rightarrow [t'](S)\]

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

\[
\begin{align*}
\text{[Int]}(S) &= \mathcal{Z} \text{ (the set of integers)} \\
\text{[Bool]}(S) &= \mathcal{T} \text{ (the set of truth values)}
\end{align*}
\]

and more interestingly

\[
\begin{align*}
[t \rightarrow t'](S) &= [t](S) \rightarrow [t'](S) \text{ (functions)} \\
[t \times t'](S) &= [t](S) \times [t'](S) \text{ (pairs of values)} \\
[t \text{ list}](S) &= [t](S)^* \text{ (sequences of values)}
\end{align*}
\]

so that \([t](S)\) 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

\[
\begin{align*}
\text{[Int]}(\mathcal{I}) &= \mathcal{Z} \\
\text{[Bool]}(\mathcal{I}) &= \mathcal{T} \\
[t \rightarrow t'](\mathcal{I}) &= [t](\mathcal{I}) \rightarrow [t'](\mathcal{I}) \\
[t \times t'](\mathcal{I}) &= [t](\mathcal{I}) \times [t'](\mathcal{I}) \\
[t \text{ list}](\mathcal{I}) &= [t](\mathcal{I})^* \\
[t \rightarrow t'](\mathcal{I}) &= \mathcal{I}_{\rightarrow t'}(\mathcal{I})
\end{align*}
\]

As an example we get \([\text{Sum}_1](S) = [\text{Sum}_2](S) = \mathcal{Z}^* \rightarrow \mathcal{Z}\).
Table 2: Fragments of the expression part of S

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 □, Tuple, Fst, etc. In Table 2 we give parts of the specification of S (using an appropriate mathematical notation in boldface). The meaning of the compile-time constructs is predetermined except for that of fix e. Here the interpretation I specifies a function I^fix_e for each type t of fix e. In Table 2 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 thereby programs. Following what we did for types we shall define a value [e](I) in [f](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

\[
\begin{align*}
\lambda x[t].e[I] env v & = [e][I] env[v/x] \\
[e(e')][I] env & = [e][I] env ([e'][I] env) \\
[x][I] env & = env(x)
\end{align*}
\]

where env[v/x] is as env except that x has the value v. The next example clauses illustrate how the interpretation I is used

\[
\begin{align*}
[e \sqcup e'][I] env & = I_{\sqcup} ([e][I] env, [e'][I] env) \\
\text{Tuple}(e,e')[I] env & = I_{\text{Tuple}}([e][I] env, [e'][I] env)
\end{align*}
\]

[Fix e][I] env = I_{fix}^e([e][I] env)

where fix e has type t

To summarize, [e][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 I (if the constructor is a combinator or fix) or we will use a standard approach.

Finally, the semantics of a program is defined to be that of the expression. The specification of S given in Table 2 is sufficient to determine the meaning of sum'0b to be

\[
\begin{align*}
[\text{sum}'0b](S) 1 & = \text{FIX}((\lambda g.\lambda (v,l).((l=[]) \rightarrow v | (l=a:l') \rightarrow a+g(v,l')))(0,l)
\end{align*}
\]

7 Forward abstract interpretation

In the previous sections we have seen some very simple program transformations that are universally applicable in the sense that whenever a given pattern matches an expression then it can safely be replaced by another 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'0b where obviously the first component of g's parameter always will have the value 0 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 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 \texttt{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 $P$

The purpose of constant propagation [1] is to determine whether an expression always will evaluate to a constant and then to determine that constant. The analysis will be specified as an interpretation $P$ following the pattern described in the previous section. So we shall define

- the meaning $P_{t ightarrow t'}$ of the run-time types $t ightarrow t'$, and
- the meanings $P_\text{int}$, $P_{\text{tuple}}$, etc of the combinators and the fixed point.

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

- $n$, an integer, meaning that the 'real' value always will be equal to $n$ (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.

This set of properties is called $Z^\top$ and similarly we can extend $T$ to $T^\top$. The type part of $P$ will now define

$$P_{\text{Int}}(P) = Z^\top$$

$$P_{\text{Real}}(P) = T^\top$$

$$[[t 	imes t']](P) = [[t]](P) \times [[t']](P)$$

$$[[t \text{ list}]](P) = (([[t]](P))^\top)^\top$$

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 $T$. This value 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 $[[t]](P)$ will contain an element (ambiguously denoted $\top$) representing that the 'real' value cannot be determined to be a constant. It will also contain an element (ambiguously denoted $\bot$) representing that the 'real' value definitely will be undefined.

As in the standard semantics we have $[\text{Sum}_1](P) = Z^\top \rightarrow Z$ whereas $[\text{Sum}_2](P) = ((Z^\top)^\top)^\top \rightarrow Z^\top$.

7.2 Constant propagation: the expression part of $P$

The expression part of $P$ will specify how to operate on these properties. In Table 3 we give a few illustrative clauses (using appropriate mathematical notation). The clauses for $\text{Snd}$, $\text{Snd}$, $\text{Const}$ and $\text{Id}$ are as in $S$. The clauses for $\text{Hd}$, $\text{T1}$, $\text{Isnil}$ and $+$ are as in $S$ but extended to cope with the special properties $T$ and $\bot$. In the clause for the conditional we distinguish between whether the test evaluates to $\text{true}$, $\text{false}$, $\top$ 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 $\text{true}$ or it may be false and we shall therefore combine the properties obtained from the two branches using the least upper bound operation $\sqcup$. For $Z^\top$, $p \sqcup q$ is defined to be $T$ if $p$ and $q$ are distinct integers whereas it is $p$ if they are equal. Furthermore $p \sqcup T = T \sqcup p = T$ and $p \sqcup \bot = \bot \sqcup p = p$ for all $p$. Finally, the specification of $P_{\text{fix}}^t$ 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 \( \mathcal{P} \) to \( \text{sum}_{9B} \) we then get

\[
\text{[\text{sum}_{9B}]}(\mathcal{P}) \ p = (p=[ ])\rightarrow 0 \ | \ (p=q;p')\rightarrow \top \n \]

\[
(p=\top)\rightarrow \top \n
\]

7.3 The enabled program transformation

We cannot deduce very much from \( \text{[\text{sum}_{9B}]}(\mathcal{P}) \) about the values arising internally in \( \text{sum}_{9B} \). We shall therefore need a sticky variant [16,19] of the analysis that will record the arguments supplied to \([e](\mathcal{P})\text{ env}\) for the various subexpressions \( e \) of \( \text{sum}_{9B} \). For the true-branch \( \text{Fst}[ ] \) of the conditional we then get that \( \text{[Fst[ ]]}(\mathcal{P})\text{ env} \) is called

with two different parameters namely \( (0,[ ]) \) (if the test evaluates to true) and \( (0,\top) \) (if the test evaluates to \( \top \)). In both cases \( \text{[Fst[ ]]}(\mathcal{P})\text{ env} \) will evaluate to \( 0 \) so we see that the analysis gives the expected result.

The constant propagation analysis enables a program transformation called constant folding [1]:

Assume that \( e \) has a subexpression \( e' \) such that during the computation of \( [e](\mathcal{P})\text{ env} \) all calls of \( [e'](\mathcal{P})\text{ env} \) return the property being the constant \( v \).

Then \( e \) can safely be changed to the expression \( e'' \) obtained by replacing the subexpression \( e' \) by \( \text{Const}[ ] v \).

Applying this transformation to \( \text{sum}_{9B} \) we get the program \( \text{sum}_{9B}^p \).

\[
\text{VAL fix(} \lambda g[ ].\text{Cond(Isnil[ ] }\square \text{ Snd[ ]}, \text{Const[ ] }0, \\
+[ ] \square \text{Tuple(Hd[ ] }\square \text{ Snd[ ]}), \text{Const[ ] }0, \text{Id[ ]}) \\
\text{g }\square \text{Tuple(Fst[ ],Tl[ ]}) \text{Snd[ ]})))) \\
\square \text{Tuple(Const[ ] }0[ ],\text{Id[ ]})
\]

\( \text{HAS Int list } = \text{Int} \)

This transformation only preserves partial correctness in the sense that the original program may loop in situations where the transformed program will not loop.

8 Backward abstract interpretation

In liveness analysis [1] 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 uses 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.
8.1 Liveness analysis: type part of \( \mathcal{L} \)

The liveness analysis will be specified by an interpretation \( \mathcal{L} \). A property of a value of type \( \text{Int} \) (or \( \text{Bool} \)) 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 \( \mathcal{L} \) will e.g. have

\[
[\text{Int}](L) = \{ \text{dead}, \text{live} \}
\]
\[
[\text{Bool}](L) = \{ \text{dead}, \text{live} \}
\]
\[
[t \times t'](L) = [t](L) \times [t'](L)
\]
\[
[t \text{ list}](L) = \{ \text{dead}, \text{live} \}
\]

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 uses of lists are defined to be \( \{ \text{dead}, \text{live} \} \) so that it will not be possible to see if only parts of the list will be needed in the future computations. If a more refined analysis is wanted then we may e.g. replace the definition above with

\[
[t \text{ list}](L) = \{ L \subseteq [t](L)^* \} \quad \text{(see e.g. [25])}.
\]

With the definition above each set \([t](L)\) will contain an element \( \bot \) (ambiguously denoted \( \text{dead} \)) representing that the 'real' value will definitely not be needed in the future computations and it will contain an element (ambiguously denoted \( \text{live} \)) representing that the 'real' value may be needed in the future computations. Because the analysis is a backward analysis the properties of functions will be functions mapping properties in the opposite direction so we have

\[
L_{t \rightarrow t'} = [t'](L) \rightarrow [t](L)
\]

Then \([\text{Sum}_1](L) = Z^* \rightarrow Z\) and \([\text{Sum}_2](L) = \{ \text{dead}, \text{live} \} \rightarrow \{ \text{dead}, \text{live} \}\.

8.2 Liveness analysis: the expression part of \( \mathcal{L} \)

Turning to the expression part of \( \mathcal{L} \) we shall specify how to operate on the properties above. In Table 4 we give a few illustrative cases. The clause for \( \Box \) reflects that functions have to be composed in the opposite direction of the standard one. In the clause for \( \text{Tuple} \) we are given a pair \( (p, q) \) of properties for the future uses of the result of \( \text{Tuple}(\cdots, \cdots) \) and we use the least upper bound operation \( \sqcup \) to join the results. On the set \( \{ \text{dead}, \text{live} \} \) the operation is defined by \( \text{dead} \sqcup \text{live} = \text{live} \sqcup \text{dead} = \text{live} \sqcup \text{live} = \text{live} \sqcup \text{dead} = \text{dead} \). (This corresponds to the least upper bound in a partially ordered set where \( \text{dead} \sqsubseteq \text{live} \).) The clauses for \( \text{Fst, Snd, Ed, Tl, Isnil, Const, Id} \) and \( + \) should be straightforward. For the conditional analysis 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 \( S \). (Technically, \( \text{dead} \) is just another name for \( \bot \).)

We can now apply \( \mathcal{L} \) to the program \( \text{sum}_{\text{BB}}^P \) and get

\[
[\text{sum}_{\text{BB}}^P](L) = p
\]

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 \([\text{sum}_{\text{BB}}^P](L) \ p\) we then get that

\[
[\text{fix}(\lambda g[]\cdots)](L) \ env
\]

only is applied to the property \( p \) and always evaluates to \( (\text{dead}, p) \). 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 \( \text{fix}(\lambda g[t_1 \times t_2 \rightarrow t_3].e') \) such that during the computation of \([e](L)\ env\) each call of \( \text{fix}(\lambda g[t_1 \times t_2 \rightarrow t_3].e')(L)\ env\) return the property \( (\text{dead}, p) \) for some \( p \).

Then \( e \) can safely be changed to \( e'' \) obtained by replacing the subexpression \( \text{fix}(\lambda g[t_1 \times t_2 \rightarrow t_3].e') \) by

\[
\text{fix}(\lambda g'[t_2 \rightarrow t_3].e'[g\sqcup \text{Snd}[t_1 \times t_2]/g]\Box \\
\text{Tuple(Fix}[]\Box(\text{Curry Snd}[t_2 \times t_1]), \\
\text{Id}[t_2])\Box \text{Snd}[t_1 \times t_2]
\]

where \( g' \) is 'fresh'.
Here Fix[ ] □ (Curry Snd[sx[t2x[t1]]) is a function that transform the argument of type t2 to an ('unneeded') value (⊥) of type t1. Applying this transformation to sum_{p} we get the program

\[
\text{VAL fix(λg'[ ]□(Curry Snd[⊥x[}) \circ \text{Const[ ] 0,}
\]
\[
\text{+[ ]□Tuple(Hd[ ]□Snd[ ],}
\]
\[
g'□Snd[ ]□Tuple(Fst[ ],}
\]
\[
\text{Tl[ ]□Snd[ ]□Tuple(Fix[ ]□(Curry Snd[ ]], Id[ ]))}
\]
\[
\text{□ Snd[ ] □ Tuple(Cond[ ] 0, Id[ ]))}
\]
\[
\text{HAS Int list \rightarrow Int}
\]

We can now apply a few algebraic transformations [2]

\[
\text{Snd[ ]□Tuple(e, e') \Rightarrow e'}
\]
\[
e □ Id[ ] \Rightarrow e
\]
\[
\text{Cond(e1,e2,e3) □ e ⇒}
\]
\[
\text{Cond(e1□e,e2□e,e3□e)}
\]
\[
\text{(Const[ ] 0) □ e ⇒ Const[ ] 0}
\]
\[
\text{Tuple(e1,e2) □ e ⇒ Tuple(e1□e,e2□e)}
\]

and the resulting program is sum_{p}^{t1}

\[
\text{VAL fix(λg'[ ]□(Cond(Isnil[ ]□Snd[ ]), Const[ ] 0,}
\]
\[
\text{+[ ]□Tuple(Hd[ ]□Snd[ ],}
\]
\[
g'□Snd[ ]□Tuple(Fst[ ],}
\]
\[
\text{Tl[ ]□Snd[ ]□Tuple(Hd[ ]□Snd[ ], Id[ ]))}
\]
\[
\text{□ Snd[ ] □ Tuple(Cond[ ] 0, Id[ ]))}
\]
\[
\text{HAS Int list \rightarrow Int}
\]

9 Conclusion

The distinction between binding times is important in the efficient implementation of programming languages. The functional programming style introduces an implicit distinction between binding times as higher-order functions typically are supplied with some but not all of their arguments. Using a binding time analysis we can make this distinction explicit and use it in the analysis and transformation on programs. The program transformations may be grouped in the following 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 in [20,11,15].

- **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 in [7] and a restricted form of the transformations described in the framework of [4].

- **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 [2,3,10].

- **Transformations on the run-time level that are enabled by certain program analyses**: The enabling analyses are expressed as abstract interpretations [19,25] and they may express properties obtained by forward or backward analyses [1]. The use of transformations en-
abled by program analyses is discussed in [16,1].

The binding time analysis, the combinator introduction and the concept of parameterized semantics have been implemented in the PSI-system constructed at Aalborg University Center [18,26]. We hope to extend the system to include facilities for specifying and applying program transformations (e.g. using the ideas of [8]).

Acknowledgement

This work has been supported by the Danish Technical Research Council and the Danish Natural Science Research Council.

References


