January 1996

A type-directed, on-line partial evaluator for a polymorphic language

Tim Sheard

Follow this and additional works at: http://digitalcommons.ohsu.edu/csetech

Recommended Citation
http://digitalcommons.ohsu.edu/csetech/78

This Article is brought to you for free and open access by OHSU Digital Commons. It has been accepted for inclusion in CSETech by an authorized administrator of OHSU Digital Commons. For more information, please contact champieu@ohsu.edu.
A Type-directed, On-line, Partial Evaluator for a Polymorphic Language

Tim Sherald
Oregon Graduate Institute of Science & Technology
P.O. Box 91000, Portland, OR 97291-1000 USA
sherald@cse.ogi.edu

The research reported in this paper was supported by the
USAF Air Materiel Command, contract # F19628-93-C-0009.
OGI Tech Report 96-004

Abstract

Recently, Olivier Danvy introduced a new, simple method for implementing powerful partial evaluators:
Type-directed partial evaluation. While type-directed partial evaluators are simple to construct, small in size, elegant, and need no analysis other than type inference, Danvy’s implementation technique had several drawbacks: it could not handle polymorphic functions, terms with free variables, the residualization of recursive functions, or the residualization of inductive datatypes.

This paper introduces a type-directed, on-line, partial evaluator, for a polymorphic lambda calculus with products, sums, and fixed points (recursion) over both values and types which fixes all these drawbacks. It introduces a novel, new implementation technique that embeds types in values, which enables these extensions. This implementation technique makes type-directed partial evaluators practical for “real” languages.

The paper also introduces a new way of thinking about type-directed partial evaluators as expansion-reduction systems. This analogy clarifies many of the subtleties inherent in type-directed partial evaluation.

1 Introduction

We have constructed a type-directed, on-line partial evaluator for a strongly typed functional language with all of the features functional programmers have come to expect, such as polymorphism, inductive datatypes, and recursion. Type-directed partial evaluation uses a simpler strategy than earlier systems, yet still provides much of the same power; including polyvariant specialization, higher order functions, partially static data structures, and automatic propagation of static contexts over dynamic branches.

Simple implementations of effective partial evaluators are important because many systems with “small” embedded languages could benefit from the use of partial evaluation technology. Until now, the effective use of a partial evaluator meant integrating one of the available, state of the art, off-the-shelf partial evaluators with the embedded language. Writing an effective partial evaluator for even a small language was beyond the reach of most embedded language implementors. This no longer need be the case.

Partial evaluation is a transformation that takes as input a (source) program plus some of that program’s inputs, and produces another (residual) program that is an efficient version of the source program specialized to the given input. It is traditionally implemented by encoding the source program and its input in a data structure, and performing a symbolic evaluation of the this data structure taking advantage of the known inputs to produce the residual program.

Type-directed partial evaluation, on the other hand, achieves similar results, only in new and novel ways. Type directed partial evaluators can be implemented by reification. A semantics provides a meaning to a syntactic program. In its purest sense, reification translates from the semantic domain back to an equivalent expression in the syntactic domain. Such a mechanism supplies the most important part of a partial evaluator. Any curried function of two or more arguments can be partially evaluated by applying it to its known argument(s) then reifying the resultant semantic value to get a syntactic representation of the specialized function as the residual program.

Our system can be explained as an expansion-reduction system. Reduction takes a term to a smaller, simpler form. Expansion takes a term to a larger, potentially more complex term. The utility of such systems is that sometimes, a tiny amount of expansion will make possible a much larger reduction, driving the term to a much simpler form. Unfettered expansion can lead to non-termination, so determining when to expand is the key to implementing effective expansion-reduction systems.

The contribution of this paper is that effective partial evaluators for realistic languages can be built as expansion-reduction systems, and that the type-directed strategy of Danvy [9] is exactly the mechanism needed to determine when expansion is necessary. The expansion used is a generalization of $\eta$-expansion.

Using this strategy the reduction mechanism is implemented as a function that maps an abstract syntax object of type $\text{exp}$ to a simpler type called a $\text{value}$. The beauty of this system is that the operational semantics of the language is the reduction part of the expansion-reduction system. Partial evaluation is implemented, in part, as a function that takes a $\text{value}$ back to an equivalent $\text{exp}$.

\footnote{No manual binding time improvements are necessary.}
We use a novel implementation technique in which types are embedded in values. A disadvantage of this technique is that type information in the form of type tags must continually be passed around by the implementation (but only at partial evaluation-time). An advantage of this technique is that every value embeds its own type, and this allows our implementation to extend the work of Danvy by handling:

- polymorphically typed functions,
- free variables in terms,
- an explicit fixed-point operator, and
- inductive datatypes.

In summary, this paper illustrates a new, simple method of implementing powerful partial evaluators for rich languages.

2 Type Based Reification

This paper explains type-directed partial evaluation by providing implementations for a sequence of lambda calculus variants of increasing complexity. In each variant we will construct a domain of types (\texttt{typ}), a syntactic domain (\texttt{exp}), and a semantic domain (\texttt{value})\(^2\).

The meaning function, \texttt{eval}, provides the reduction engine of the expansion-reduction system, as well as the operational semantics of each calculus. Two mutually recursive (type-directed) functions \texttt{reify} and \texttt{reflect} aid in the implementation of partial evaluation. The function \texttt{reify} maps from values back to equivalent \texttt{exp}s. This is possible because our implementation expands the notion of a value to include additional information, such as types and a mechanism for embedding syntactic terms in semantic values. The function \texttt{reflect} implements the expansion portion of the expansion-reduction system, mapping from simple \texttt{exp}s to more complex \texttt{values}.

Our first calculus is a simply typed lambda calculus with integer constants. Danvy describes a type-directed partial evaluator for this calculus where the domain of syntax is Scheme, the meaning function is the Scheme compiler, and the reification algorithm is type directed, having an explicit type parameter. In Figure 1 we reproduce this work by exhibiting an SML implementation for the first lambda calculus in our sequence.

In this calculus there are only integer types (\texttt{tint}) and function types (\texttt{tarrow}) (though other base types other than integer would work just as well). There are four kinds of elements in the syntactic domain: integer constants (\texttt{eint}), function application (\texttt{eapp}), abstractions or function construction (\texttt{eabs}), and variables (\texttt{evar}).

The domain of values contains constructors for integers (\texttt{vint}) and functions (\texttt{vfun}) corresponding to the types \texttt{tint} and \texttt{tarrow}. In addition, the domain of values must also contain the coercion\[^2\] \texttt{vdyn}. This constructor is crucial to implementing reification in a typed language such as SML. It allows the syntactic domain to be embedded in the semantic domain. Dynamic values are transparent at the programmer level interface and are used only by the reification process.

The evaluation function is completely standard. It uses an environment, \texttt{env}, to map variables to values and has type: \texttt{env -> exp -> value}. It uses two auxiliary functions \texttt{getV} to apply the environment to a variable, and \texttt{extV} to extend an environment at a particular variable to a new value.

In this calculus reification works over values produced by \texttt{eval} from closed terms, and is implemented by the mutually recursive functions \texttt{reify} and \texttt{reflect}.

2.1 How does Reification Work?

The functions \texttt{reify} and \texttt{reflect} are used to map values back to expressions. Reifying an integer value is trivial. Reifying a function is more difficult. We must somehow build an \texttt{exp} when all we have is a function from \texttt{value} to \texttt{value} which implements it's be-

---

\[^2\]We adhere to the convention that the initial letter of each constructor function is one of the letters: \texttt{t}, \texttt{e}, or \texttt{v} indicating to which one of these domains the constructors belongs: \texttt{t} for types, \texttt{e} for expressions (the syntactic domain), and \texttt{v} for values (the semantic domain).
behavior. As strange as this may seem, if the function was produced by `eval` from a closed term then this is possible[8, 2].

Correctly reifying such a function can be done if one knows the type of the function; this is why both `reify` and `reflect` take an initial `typ` parameter.

A key to this process is the `reflect` function. A `value to value` function, `f`, expects an argument with a particular shape; i.e. constructed in a particular way from the constructors `vint` and `vfun`. If it is applied to a value with the wrong shape an error will occur. Normally type correctness filters out programs in which such errors may occur, but in reification we may apply `f` to a `vdyn` value. We must `expand` such a dynamic value to have the additional shape `f` demands. Without this expansion we cannot proceed, since application of the function `f` could fail. Expansion at precisely this place is the key to making this strategy work. The types indicate exactly how the expansion is to proceed.

To reify `vfun(f)`, with type `tarrow(t1,t2)`, an abstraction whose bound variable is a new fresh variable `s` is constructed. This variable is expanded, by `reflect` into a value with the proper shape. The “semantic” function `f` is applied to this dynamic value and the function `reify` is applied to the result to obtain the body of the abstraction.

If `vfun(f)` was constructed by `eval` from a closed term then all `f` can do is “push its argument around”. Thus if the dynamic value constructed has the correct shape then nothing can go wrong. For example consider the identity function at the type `tarrow(tint,tint):

\[\text{let } \text{fun } F \text{ v } = \text{eval (extV x v env)} \text{ body in } vfun(t,F) \text{ end}\]

```
fun reify (vint n) = eint n
| reify (vfun(t,f)) =
  let val s = gensym "x"
      fun g x = reify (f (reflect t x))
      in eabs(s,g (evar s)) end
| reify (vdyn e) = e

and reflect (tarrow(t1,t2)) e =
  vfun(t,fn v =>
        reflect t (eapp(e,reify v)))
| reflect (tint) e = vdyn e
```

Figure 2: Version with Embedded Types

The extra information necessary is the type of the domain of a `vfun` value. Since `vfun` values are constructed from `eabs` expressions by `eval`, we will need to annotate `eabs`, with type information as well. This is not an undue burden because the concrete syntax does not carry this information; instead it can be added to the abstract syntax by a type inference mechanism. In addition when we move from the simply-typed to the polymorphically-typed lambda calculus we will need this information anyway.

These changes are outlined in Figure 2. In this figure the dots (...) indicate elided segments which are identical to those in Figure 1. Note the additional `typ` component of the constructors `eabs` and `vfun`, and how this type is propagated into `vfun` values by the `eval` function and used by `reify` to direct the expansion process by being passed as the parameter to `reflect`.

Note that reification is still type-directed, even though a `typ` is no longer an explicit parameter to `reify`.

3 An Important Contribution of this Work

An interesting observation about the behavior of the function `reify` leads to a significant improvement in the implementation of reification based partial-evaluators by enabling extensions that fix the drawbacks of Danvy's original paper.

Note that the structure of values (i.e. the different constructors `vint` and `vfun`) contain partial information about the types of a value. By embedding a small, additional amount of type information in `vfun` values it is possible to provide all the type information necessary for reification completely internal to a value. This change in the implementation allows the function `reify` to no longer need a separate `typ` input, and directly supports the extensions that follow.

```
datatype exp = ... | eabs of typ * string * exp

datatype value = ... | vfun of typ * (value -> value)

fun eval env e =
  case e of ...
| (eabs(t,x,body)) =>
    let fun F v = eval (extV x v env) body
    in vfun(t,F) end

fun reify (vint n) = eint n
| reify (vfun(t,f)) =
  let val s = gensym "x"
      fun g x = reify (f (reflect t x))
      in eabs(s,g (evar s)) end
| reify (vdyn e) = e

and reflect (tarrow(t1,t2)) e =
  vfun(t,fn v =>
        reflect t (eapp(e,reify v)))
| reflect (tint) e = vdyn e
```

Figure 2: Version with Embedded Types

The extra information necessary is the type of the domain of a `vfun` value. Since `vfun` values are constructed from `eabs` expressions by `eval`, we will need to annotate `eabs`, with type information as well. This is not an undue burden because the concrete syntax does not carry this information; instead it can be added to the abstract syntax by a type inference mechanism. In addition when we move from the simply-typed to the polymorphically-typed lambda calculus we will need this information anyway.

These changes are outlined in Figure 2. In this figure the dots (...) indicate elided segments which are identical to those in Figure 1. Note the additional `typ` component of the constructors `eabs` and `vfun`, and how this type is propagated into `vfun` values by the `eval` function and used by `reify` to direct the expansion process by being passed as the parameter to `reflect`.

Note that reification is still type-directed, even though a `typ` is no longer an explicit parameter to `reify`.

4 Adding Products and Sums

Finally, products and sums are also handled by Danvy’s system. Incorporating them into our implementation is straightforward. We need only to ensure that our extensions for products and sums enforce the invariant that values embed their own types.

In Figure 3 we provide the datatype declarations necessary for adding sums and products. Products are im-
implemented as a pair of values. Sums are implemented by tagging a value with one of the constructors left or right of the tag datatype.

In the syntactic domain we must provide abstract syntax for both the introduction and elimination of products and sums. Here we explain our choice of abstract syntax by appealing to a (hopefully) familiar (though imaginary) concrete syntax which has a direct mapping into the constructors we have added to the datatype exp.

The introduction of products (epair) uses the traditional parentheses comma notation to pair two expressions, e.g. \((4, x)\). The elimination of products (esabs) uses a pattern matching abstraction with two bound variables with an explicit annotation denoting the domain of the abstraction. For example one might write: \(\langle fn\ (t_1 \ast t_2)\ (x, y) \Rightarrow x\rangle\) for the first projection function. Here \((t_1 \ast t_2)\) is the type annotation, \((x, y)\) indicate the bound variables, and the body of the abstraction is just \(x\).

Sums are introduced (esum) by tagging an expression with one of the sum injection tags. For example \((\text{left} \ x)\) or \((\text{right} \ 5)\). The elimination of sums (esabs) uses a pattern matching abstraction with two clauses. For example: \(\langle fn\ (t_1 + t_2)\ \text{left} \ x \Rightarrow 3\ | \ \text{right} \ y \Rightarrow 3\rangle\) denotes the constant \(3\) function. Note again, the explicit annotation, \((t_1 + t_2)\), denoting the domain of the abstraction.

In Figure 4 we give the semantic evaluation function eval and the reification functions reify and reflect. Again the dots (\(\ldots\)) indicate elided clauses which are identical to those in previous figures. The evaluation function is again completely standard. It is interesting to note, though, that the evaluation of product elimination (epabs) and sum-elimination (esabs) build ML functions (F and G), which use the pattern matching capabilities of the underlying implementation language to decompose their value arguments. Note that these functions will cause an SML match exception if they are applied to non-pairs or non-sums. This is one place where the errors discussed in Section 2.1 originate. In well-typed programs, without reification, this will never occur. The expansion properties of reflect ensure that these errors do not occur while reifying.

In Figure 4 the reify function over a function value \((\text{vfun}(t, f))\) must construct one of the three kinds of abstractions, i.e. product elimination, sum elimination, or ordinary lambda abstraction. By inspecting the domain this choice can be made. It builds an appropriate abstraction with “fresh” bound variables, and uses Fillinski’s reset control operator[1, 13] to delimit a new dynamic context. This context may be abstracted by a shift control operator in the reflect function.

The function reflect expands its exp argument to have the shape of its typ argument. For a product it returns a vpair where the components are the first and second projection functions applied to \(e\). Here

\[
\begin{align*}
f\text{m effst} \ t \ e & = \text{eapp}(\text{epabs}(\langle x, y \rangle, t, \text{evar} \langle x \rangle), e) \\
f\text{m emd} \ t \ e & = \text{eapp}(\text{esabs}(\langle x, y \rangle, t, \text{evar} \langle y \rangle), e)
\end{align*}
\]

Functions are expanded using eta-expansion, and reflect over a sum constructs a case statement which captures the current delimited context using shift in the abstracted variable \(x\), and pushes this context into the clauses of a case expression. Note that a case expression over a term \(e\) is simply syntactic sugar for application of a sum abstraction to \(e\). I.e. \(\text{case} \ x \ of \ C\ x \Rightarrow e \ | \ D\ y \Rightarrow f\) is the same as \((\text{fn} \ C \ x \Rightarrow e \ | \ D\ y \Rightarrow f)\ x\).

At this point we have completed an implementation of the same material found in Danvy’s paper with which he implemented using Scheme. The only difference is that we use a novel implementation technique in which types are embedded in values. A disadvantage of this technique is that type information in the form of type tags must continually be passed around. An advantage of this technique is that every value embeds its type. This allows our implementation to handle several important extensions.

5 Adding Polymorphism

Polymorphism is an important feature used extensively in modern functional languages.

To add polymorphism to our partial evaluator we need to add universally quantified types in the domain of types, and type abstraction and type application in the syntactic domain. In the semantic domain we need to add type functions which when given a type return a value specialized to that type.

\[
\begin{align*}
datatype\ \text{typ} & = \ldots \\
& | \text{tpair} \ of \ \text{typ} \ast \ \text{typ} \\
& | \text{tsum} \ of \ \text{typ} \ast \ \text{typ}
\end{align*}
\]

Figure 3: Type Additions for Sums & Products

Figure 3: Type Additions for Sums & Products

\[
\begin{align*}
datatype\ \text{tag} & = \text{left} \ | \ \text{right}; \\
datatype\ \text{exp} & = \ldots \\
& | \text{epair} \ of \ \text{exp} \ast \ \text{exp} \\
& | \text{esabs} \ of \ \text{string} \ast \ \text{string} \ast \ \text{typ} \ast \ \text{exp} \\
& | \text{esum} \ of \ \text{typ} \ast \ \text{exp} \\
& | \text{esabs} \ of \ \text{typ} \ast \ \text{string} \ast \ \text{exp} \ast \ \text{string} \ast \ \text{exp};
\end{align*}
\]

\[
\begin{align*}
datatype\ \text{value} & = \ldots \\
& | \text{vpair} \ of \ \text{value} \ast \ \text{value} \\
& | \text{vsun} \ of \ \text{tag} \ast \ \text{value};
\end{align*}
\]

The function reflect expands its exp argument to have the shape of its typ argument. For a product it returns a vpair where the components are the first and second projection functions applied to \(e\). Here

\[
\begin{align*}
f\text{m effst} \ t \ e & = \text{eapp}(\text{epabs}(\langle x, y \rangle, t, \text{evar} \langle x \rangle), e) \\
f\text{m emd} \ t \ e & = \text{eapp}(\text{esabs}(\langle x, y \rangle, t, \text{evar} \langle y \rangle), e)
\end{align*}
\]

Functions are expanded using eta-expansion, and reflect over a sum constructs a case statement which captures the current delimited context using shift in the abstracted variable \(x\), and pushes this context into the clauses of a case expression. Note that a case expression over a term \(e\) is simply syntactic sugar for application of a sum abstraction to \(e\). I.e. \(\text{case} \ x \ of \ C\ x \Rightarrow e \ | \ D\ y \Rightarrow f\) is the same as \((\text{fn} \ C \ x \Rightarrow e \ | \ D\ y \Rightarrow f)\ x\).

At this point we have completed an implementation of the same material found in Danvy’s paper with which he implemented using Scheme. The only difference is that we use a novel implementation technique in which types are embedded in values. A disadvantage of this technique is that type information in the form of type tags must continually be passed around. An advantage of this technique is that every value embeds its type. This allows our implementation to handle several important extensions.

5 Adding Polymorphism

Polymorphism is an important feature used extensively in modern functional languages.

To add polymorphism to our partial evaluator we need to add universally quantified types in the domain of types, and type abstraction and type application in the syntactic domain. In the semantic domain we need to add type functions which when given a type return a value specialized to that type.

\[
\begin{align*}
datatype\ \text{typ} & = \ldots \\
& | \text{tuniv} \ of \ \text{string} \ast \ \text{typ} \\
& | \text{tvar} \ of \ \text{string}
\end{align*}
\]

Figure 3: Type Additions for Sums & Products

Figure 5: Type Additions for Polymorphism

\[
\begin{align*}
datatype\ \text{exp} & = \ldots \\
& | \text{etapp} \ of \ \text{exp} \ast \ \text{typ} \ (\ast \text{Ex}: \ \text{len} \ [\text{int}] \ast) \\
& | \text{esabs} \ of \ \text{string} \ast \ \text{exp}; \\
& (\ast \text{Ex}: \ \text{Fn} \ \text{alpha} \Rightarrow \text{fn} \ (\text{alpha}) \ x \Rightarrow x \ast)
\end{align*}
\]

Figure 5: Type Additions for Polymorphism

\[
\begin{align*}
datatype\ \text{value} & = \ldots \\
& | \text{vtypfun} \ of \ \text{typ} \Rightarrow \ \text{value};
\end{align*}
\]
fun eval env e = case e of
  epair(x,y) => vpair(eval env x, eval env y)
| epabs(x,y,t,body) =>
    let fun F (vpair(v1,v2)) = eval (extV x v1 (extV y v2 env)) body
    in vfun(t,F) end
| esum(tg,e) => vsum(tg, eval env e)
| esabs(t,x,e1,y,e2) => let fun G (vsum(left,v)) = eval (extV x v env) e1
  | G (vsum(right,v)) = eval (extV y v env) e2
    in vfun(t,G) end
| ...;

fun reify (vpair(a,b)) = epair(reify a, reify b)
| reify (vsum(tg,v)) = esum(tg, reify v)
| reify (vfun(t,f)) =
  (case t of
tpair(t1,t2) =>
    let val s1 = gensym "y"  val s2 = gensym "z"
    val p = epair(evar s1, evar s2)
    in epabs(s1,s2,t,reset (fn () => reify (f (reflect t p)))) end
  | tsum(t1,t2) =>
    let val s1 = gensym "m"  val s2 = gensym "n"
    fun clause tag t s () = reify (f (vsum(tag,reflect t (evar s))))
    in esabs(s1,s2,reset (clause left t1 s1),s2,reset (clause right t2 s2)) end
  | _ =>
    let val s = gensym "x"
    in eabs(s,t,reset (fn () => reify (f (reflect t (evar s))))) end)
| reify ...

and reflect (t as tpair(t1,t2)) e =
  vpair(reflect(t1 (esct t e)),reflect t2 (esnd t e))
| reflect (t as tsum(t1,t2)) e =
  let val s1 = gensym "m"  val s2 = gensym "n"
  let fun clause k tag t s () = k(vsum(tag,reflect t (evar s)))
  in shift (fn k => eapp(esabs(t,s1,reset (clause k left t1 s1),
    s2,reset (clause k right t2 s2))
  )
  end
| reflect ...;

Figure 4: Additions to Eval, Reify and Reflect for Sums and Products

The evaluation function becomes more complex because the environment must now map both value variables to values and type variables to types. The type mapping is extended when evaluating type abstractions and it is used to perform type substitution on the types in type application as well as the explicit domain type annotations present in abstractions. Figure 6 provides the details of this process. Here extT extends the type mapping in the environment, and getT uses the type environment to instantiate the type variables over its typ argument.

All the difficulties in handling polymorphism are inherent in constructing values with the correct type. The eval function does this quite elegantly. Only the cases which deal with types differ from our previous version. When constructing a function value vfun(t,f), we must be sure that t is fully instantiated using the type mapping environment. This ensures that the embedded types in vfun values correctly describe the type of their function (f) counterparts. Note that this implies that every vfun function ever created is monomorphic. Type instantiation must also be done when an expression is specialized using type application.

Reification for a polymorphic language where the types are explicitly contained in the values becomes almost trivial. Reification of a type function is a type abstraction in the syntactic domain. Reflection over a term with a universal type constructs a type function, but we must first replace all occurrences of the universally bound type variable with the argument of the type function. reflect expands a type variable in the same way it expands a base type, by using the vdyn injection. This is always safe because if the function is truly polymorphic then it makes no assumption about the “shape” of its argument, and will thus never “probe” a value’s structure in a way that will cause an error.

6 Handling Non-Closed Terms

One of the restrictions in Danvy’s work was the inability to handle terms with free variables. In the imple-
fun eval env e /
  case e of /
    | (eabs(t,x,body)) ->
      let fun F v = eval (extV x v env) body
      in vfun/(getT t env/,F/) end
    | (epabs(t,x,y,body)) =
      let fun F (vpair(v1,v2)) = eval
        (extV x v1 (extV y v2 env)) body
      in vfun/(getT t env/,F/) end
    | (esabs(t,x,e1,y,e2)) =
      let fun F (vsum(left,v)) = eval
        (extV x v env) e1
      | F (vsum(right,v)) = eval
        (extV y v env) e2
      in vfun/(getT t env/,F/) end
    | (etabs(s,e)) =
      vtypfun/(fn t => eval (extT s t env) e)
    | (etapp(e,t)) =
      (case eval env e of
        vtypfun f => f (getT t env))
    | _ => vdyn e

fun reify x =
  case x of /
    | (vtypfun f) =
      let val t = gensym "t"
      in etabs(t,reify f (tvar t)) end
    | (tuniv(s,t)) e =
      vtypfun(fnt => reflect (typsub (s,[(s,t2)]) t) e)
    | (tvar _) e = vdyn e

and reflect t e =
  case t of /
    | (tuniv(s,t)) e =
      vtypfun(fn t2 =>
        reflect (typsub (s,[(s,t2)]) t) e)
    | (tvar _) e = vdyn e

Figure 6: Polymorphic Eval, Reify & Reflect

Practical systems supply primitive operations on base types. These operations usually appear as additional constructs in the syntactic domain, or as constants in the initial environment. Because reification may introduce dynamic values (constructed with vdyn) every primitive function needs to know how to react when it is applied to such a value.

Consider an addition function present in the initial environment. It might be encoded as a function value as follows:

\[
\begin{align*}
\text{fn reify x =} & \\
\text{  case x of } & \\
\text{    | (vtypfun f) =} & \\
\text{      let val t = gensym "t"} & \\
\text{      in etabs(t,reify f (tvar t)) end} & \\
\text{    | (tuniv(s,t)) e =} & \\
\text{      vtypfun(fnt => reflect (typsub (s,[(s,t2)]) t) e)} & \\
\text{    | (tvar _) e = vdyn e} & \\
\end{align*}
\]

Because one of the arguments to the primitive \texttt{inteq} is dynamic, the application:

\[
\text{eval(evapp(x,y)) =}
\]

\[
\text{(case (eval env x) of}
\]

\[
\text{  vfun(_,f) => f (eval env y))}
\]

This clause is replaced with an application to a new \texttt{App} function which checks for and handles this contingency:

\[
\begin{align*}
\text{fun App (vfun(t,g),vdyn e) = g (reflect t e)} & \\
\text{  | App (vfun(_,g),x) = g x} & \\
\text{  | App (vdyn e,x) = vdyn (evapp(e,reify x))} & \\
\end{align*}
\]

This function is smart. When it is applied to a dynamic argument, it knows how to reconstruct its syntactic representation. This solution is present in Danvy's paper, but can cause additional problems which must be addressed.

7 Handling Primitives and Constants

Without primitives, functions created by \texttt{eval}, in a type correct program, are never applied to values with the wrong shape. In the reification process, the use of the \texttt{reflect} expansion guarantees that enough “structure” is wrapped around \texttt{vdyn} values to make them “invisible” to the functions which are applied to them. When primitives propagate dynamic values, this is no longer the case. Now, functions created by \texttt{eval} must also be designed to handle syntactic, dynamic values as well as ordinary semantic values. Consider the expression:

\[
\begin{align*}
\text{fn x =>} & \\
\text{  case (inteq(x,3)) of True => 5 | False => 6} & \\
\end{align*}
\]

Here, \texttt{inteq} is a smart primitive as outlined above, and \texttt{True} and \texttt{False} are shorthands for elements of the type \texttt{int int} used to denote booleans, namely: \texttt{True = left 0} and \texttt{False = right 0}. The expression above has the following abstract syntax:

\[
\begin{align*}
\text{eabs(tint,"x"),} & \\
\text{evapp(esabs(tsum(tint,tint),} & \\
\text{  "y",eint 5,"z",eint 6),} & \\
\text{evapp(evar "inteq",} & \\
\text{  epair(evar "x",eint 3))}) & \\
\end{align*}
\]

Because one of the arguments to the primitive \texttt{inteq} is dynamic, the application:

\[
\text{evapp(evar "inteq",evapp(evar "x",eint 3))}
\]

returns a dynamic value. This causes the the sum abstraction to be applied to a dynamic value. But functions created from sum abstractions (see the \texttt{esabs} clause of \texttt{eval} in Figure 4) do not handle dynamic values.

The evaluation function must be modified so that when a function value is applied to a dynamic value we reflect over this value to give it the shape that the function expects. In \texttt{eval}, functions are applied in only one place, in the clause for application (\texttt{evapp}).

\[
\begin{align*}
\text{... eval env (evapp(x,y)) =} & \\
\text{  (case (eval env x) of} & \\
\text{    vfun(_,f) => f (eval env y))} & \\
\end{align*}
\]

This clause is replaced with an application to a new \texttt{App} function which checks for and handles this contingency:

\[
\begin{align*}
\text{... eval env (evapp(x,y)) =} & \\
\text{  App(eval env x, eval env y)} & \\
\end{align*}
\]

\[
\begin{align*}
\text{fun App (vfun(t,g),vdyn e) = g (reflect t e)} & \\
\text{  | App (vfun(_,g),x) = g x} & \\
\text{  | App (vdyn e,x) = vdyn (evapp(e,reify x))} & \\
\end{align*}
\]
It is also possible that the function part of an application can be a dynamic value. The `App` function also handles this by reifying the argument and constructing a dynamic application.

The ability to opportunistically expand dynamic values which are the arguments of function application is crucial to handling primitive functions in systematic way. We consider this a second important contribution of paper. With out this ability, a system is forced to either do without primitives, drastically reducing its utility, or to handling them in a non-uniform and ad-hoc manner.

### 7.2 A Subtle Distinction

Is it always necessary to reflect over a dynamic argument before applying a function value? The answer to this question is no, but the reasoning necessary to answer it is quite subtle. If the function was constructed by `eval` then the reification must be performed, otherwise an SML match exception may occur. But, if the function is a smart primitive, then reification is not necessary as such a function is designed to handle dynamic values. Thus, the correct action depends upon being able to distinguish primitive functions from ordinary functions. In fact, being able to distinguish between functions by their origin, will be essential for our treatment of recursive functions as well.

To make this choice `vfun` values must be tagged to distinguish their ability to handle dynamic values.

```plaintext
datatype IQ = dumb | smart;

datatype value = ...
  | vfun of IQ * typ * (value -> value)

fun eval env e =
  case e of ...
    | (eapp(x,y)) => App(eval env x,eval env y)

and App (f,x) =
  case (f,x) of ...
    | (vfun(dumb,t,g),vdyn e) => g (reflect t e)
    | (vfun(_,t,g),x) => g x
    | (vdyn e,x) => vdyn(Eapp(e,reify x))
    | (a,b) => vdyn(Eapp (reify a,reify b))

and reify v =
  case v of ...
    | reify (vfun(smart,t,f)) =
      let val s = gensym "x"
      in eabs(t,s,reset
        (fn () => reify (f (vdyn (evar s)))))
      end

and reflect t e = ...
```

Figure 7: Smart Function Application

In Figure 7 an implementation of this is provided. A new datatype, `IQ`, is added which is used to tag `vfun` values. The `App` function uses this information to choose whether or not a dynamic value needs to be reflected before function application.

One other optimization is now possible: when reifying a `smart` function it is no longer necessary to reflect over the fresh newly bound variable to construct a value with the proper shape. Smart functions are designed to handle dynamic values. The clause for smart functions in `reify` is added to take advantage of this fact.

### 8 Recursive Functions

Handling recursive functions is problematic for partial evaluators. Should a recursive function be unfolded, or should it be specialized and then residualized? In a type-directed partial evaluator unfolding and specialization are handled simply by application followed by reification. But, what if a recursive function needs to be residualized because the recursion is controlled by dynamic arguments?

In our system we have a simple solution that works some of the time: a recursive function should be specialized if it is applied to a dynamic value, otherwise it is unfolded. This has turned out to be quite effective in the programs we used our system on, but is also far from optimal.

We accomplish this by introducing an explicit fixed-point combinator into the syntactic domain. A concrete syntax using this combinator is: 

```
Y f => fn x => e
```

This is equivalent to: `let fun f x = e in f end`. But we prefer the former as it is closer to the abstract syntax introduced in Figure 8. Because the language is given a strict semantics, `Y` must only be used to construct functions. The evaluation mechanism constructs a `vfun` value from a `Y` expression.

The key idea to handling recursion is to capitalize on the idea that values should carry additional information. Previously we argued that values should carry type information, and that functions values should carry information indicating their source. We will extend this idea by making recursive functions “smart” by endowing them with the ability to residualize themselves. This can be seen from the implementation, in that the recursive functions returned by `eval` when operating on a `Y` combinator have an additional clause indicating how they behave when applied to a dynamic argument.

In Figure 8 the syntactic combinator `ey(t,s,e)` is introduced, which represents `fix(λ s : t . e)`. It binds `s` inside `e` to the value of the whole expression, which is also returned as its value.

The `eval` function gives meaning to this expression by tying a recursive knot using the recursive function capability of the meta language ML, to define a function `F` which is then embedded in a `vfun` value. The first clause of `F` defines how it behaves on values and is described later. The second clause of `F` gives meaning to the `ey` combinator. `F` is defined in terms of the evaluation of the `body` of the combinator in an extended environment which binds the variable, `x` of the combinator to a `vfun` which embeds `F`. The body is evaluated, this should return a function, which is then applied to the argument `v`. Thus the recursive knot is tied.

The first clause makes `F` `smart`. `F` reconstructs itself if applied to a dynamic value. This is done by con-
datatype exp =
  | ey of typ * string * exp (* Y s => e *)

fun eval env e =
case e of
  | ey(t as tarrow(tarrow(d,r),_),x,body) =>
    let
      fun F (vdyn e) =
        let val s' = gensym x
        val body' = reify (eval (extV x (vdyn (evar s')) env) body)
        in vfun(dumb, getT d env, F) end
    in vfun(smart, getT d env, F) end
  | F v = Apply(eval (extV x (vfun(smart, getT d env, F))) body, v)

and reify (vfun(_,t,f)) =
case t of
  | tmu(s,t1) =>
    let val s1 = gensym "y"
    in einabs(t,s1, reset (fn () =>
      reify (f (vin (reflect t1 (evar s1))))) end
  | tmu(_,_) =>
    let val z = gensym "z"
    in Eapp(einabs(t,z,evar z),e) end

Figure 8: Recursive Functions

Structuring a new ey term with a fresh bound variable s'. The body of this term is the old body evaluated under a new environment binding x to s'. We can illustrate this with the following example. It applies a recursive implementation of addition using increment and decrement to a constant 5.

(Y F =>
  fn x =>
    fn y =>
      case inteq(x,0) of
      | True => y
      | False => plus(1,F (minus(x,1)) y))
)

5

The evaluation of the Y combinator returns a smart function. Because this is applied to a real value (vint 5), the function unfolds itself until x is equal to 0. Since y is dynamic the smart plus function reconstructs itself returning a dynamic value for each recursive call, finally returning the dynamic value: (fn(int) x =>
  plus(1,plus(1,plus(1,plus(1,plus(1,x)))))).

Note that this implementation of recursive functions can cause evaluation under a lambda, and in general this is unsafe as it may lead to non-termination. See section 12 for details.

9 Inductive Types

Inductive types are problematic for similar though slightly different reasons than recursive functions are problematic. They potentially cause infinite unfoldings. Such infinite unfoldings will in the reflect function when an expression with an inductive type is coerced into a value with an inductive “shape”. The value constructed will be infinite. Our solution to this is to reflect an expression into a value with only one level of unfolding over an inductive type, and to rely upon the smart reconstruction abilities of recursive functions in general (as outlined above) to drive partial evaluation of recursive functions over inductive types.

datatype typ =
  | tmu of string * typ;

datatype value =
  | vin of value;

datatype exp =
  | ein of exp
  | einabs of typ * string * exp;

fun eval env e =
case e of
  | ein x => vin(eval env x)
  | einabs(t,x,body) =>
    let fun F (vin v) =
      eval (extV x v env) body
    in vfun(dumb, getT t env, F) end
and reify (vfun(_,t,f)) =
case t of
  | tmu(s,t1) =>
    let val s1 = gensym "y"
    in einabs(t,s1, reset (fn () =>
      reify (f (vin (reflect t1 (evar s1))))) end
  | tmu(_,_) =>
    let val z = gensym "z"
    in Eapp(einabs(t,z,evar z),e) end

Figure 9: Extensions for Inductive Types

Figure 9 introduces the additional machinery necessary to handle inductive types. A new type constructor, tmu, used to construct inductive types is added to typ. Two additional operators in the syntactic domain are added: ein to introduce values of an inductive type and einabs to eliminate them. The syntax (fn x => e) creates a function whose domain is (tmu s => T s), and which binds x with type T(Mu s => T s) inside e. The evaluation of these syntax constructs is straightforward. Evaluation either adds or subtracts the new semantic constructor vin.

For example some functions over lists of integers could be defined as follows:

type IntList = Mu x => unit + (int * x);
val nil = In (left ());
val cons = fn (x, y) => In (right (x, y));
val hd = fn (In x) =>
  case x of left x => error | right (a, b) => a;

Reification of a function with an inductive domain introduces an inductive abstraction (einabs). A fresh variable $s$ with type $T(Mu s => T s)$ is reflected over to obtain a value. This value is then injected into the type $Mu s => T s$ by the $\text{win}$ constructor, $f$ is then applied to this value and the result reified to obtain the body of the abstraction.

Rejection of an expression, $e$, over inductive type $tmu(s, t1)$ creates a value with only one level of the unwinding. First $e$ is projected out of the inductive domain using $\text{cout}$ and then the body $t1$ of the $tmu$ is used.

9.1 Nested Patterns and Lazy Expansion

What happens if we have a function over an inductive type that is non-recursive, but because of the use of nested patterns, demands more than one level of unrolling in the expansion phase? Consider the $\text{cadr}$ function below with type list of integer to integer, which returns the head of the tail of a list if it has at least two elements and returns zero otherwise.

$$\text{fun cadr (In xs)} =$$
$$\text{case xs of}$$
$$\text{Nil () => 0}$$
$$\text{| Cons(y, In ys) =>}$$
$$\text{(case ys of Nil () => 0 | Cons(z, zs) => z);}$$

Rejection of $\text{cadr}$ causes the reflection of the abstraction variable at an inductive type. Unrolling this type just once leaves the tail of the list as a dynamic variable. Applying $\text{cadr}$ to this single unrolling will fail when the inner case is applied to this dynamic variable. Fortunately, the strategy of Section 7.2 saves the day. The case is a dumb function; so when it is applied to a dynamic variable, application reflects this variable forcing another level of unrolling. If a recursive function forced this kind of lazy unrolling, non-termination would result, but because recursive functions are smart this is not a problem.

10 Post Processing

Reification is quite general, but because it is type-directed the reification of very simple functions such as the identity function and constant functions is somewhat more complicated than it need be. For example the reification of the constant 5 function at type (int+int) -> int is:

$$(\text{fn (int+int)} \text{ left x => 5 | right y => 5})$$

rather than the simpler $$(\text{fn x => 5})$$. We have found that a simple post processing phase after reification has been completed makes the reified code much more presentable. The following transformations rules are quite useful: an explicit product rewrites from $$(\text{fn (x, y) => x})$$. The other transformation we find useful is the rewriting of an abstraction over a sum where each clause of the sum-abstraction contains an identical context where the hole in the context is a tagging of the pattern variable. For example: $$(\text{fn (ti+ti2) left x => f (left x) | right y => f(right y)})$$ rewrites to $$(\text{fn x => f x})$$. Experience may show that additional rules are also useful.

11 An Example

We have built a type-directed partial evaluator implementation based upon the ideas outlined above. Our implementation is for a richer language including n-ary products and n-ary sums. Our system uses a simple read-eval-print loop interface, and incorporates a Hindley-Milner type inference system which automatically adds the necessary type annotations. The partial evaluator can handle a richer type system than the Hindley-Milner type inference engine actually constructs.

As an illustration of the power of such a system we have included the complete code of an example in Figure 10 in the appendix. This example implements a rewrite system over an expression language. A rule has the form $lhs \Rightarrow rhs$ where all the variables in $rhs$ must appear in $lhs$. For example the rule $$(x + y) + z \Rightarrow x + (y + z)$$

specifies a program which when applied to a subject term of the form $$(x + y) + z$$ returns $x + (y + z)$. If the subject term does not have this shape the subject term is returned unchanged.

In the appendix such a system is implemented in a completely naive manner. Rewriting is implemented by a function $\text{rewrite}$ which takes a pattern, a subject term and returns the (possibly) transformed subject term. It decomposes the pattern into left and right-hand sides and uses the function $\text{match}$ to build a substitution from the left-hand side and the subject term; if successful, it applies the substitution to the right-hand side of the rule.

The substitution is computed by the function $\text{match}$. It performs a simultaneous recursive walk over a pattern term and a subject term, returning a substitution which pairs the variables in the pattern to the matching subterms of the subject term. If at any point the subject term fails to have the “shape” of the pattern the failure substitution is returned.

When $\text{rewrite}$ is partially applied to a rule specifying $$(x + y) + z \Rightarrow x + (y + z)$$ a function from term to term is returned. When reified this function returns the following residual program:

$$\text{fn d1 =>}$$
$$\text{case out d1 of}$$
$$\text{Var d4 => In (Var d4)}$$
$$\text{| Op (a, b, c) =>}$$
$$\text{if streq(b,"+")}$$
$$\text{then}$$
$$\text{case out a of}$$
$$\text{Var d8 => In (Op (a, b, c))}$$
$$\text{| Op (x,y,z) =>}$$
$$\text{if streq(y,"+")}$$
$$\text{then In (Op ((x,"+"), In (Op (z,"+"), c))))}$$
$$\text{else In (Op (a, b, c))}$$
$$\text{| Int d2 => In (Op (a, b, c))}$$
$$\text{| Int d41 => In (Int d41)}$$

The reification has completely reduced all static computations. It has performed the matching against the pattern (left-hand side) part of the rule and the substitution of the right-hand side. The same program using a
more traditional partial evaluator often requires a binding time improvement, typically making the application of the substitution an explicit continuation parameter of \texttt{match}, making it harder to both understand and maintain.

This example illustrates the usefulness of the extensions to the earlier work on type-directed partial evaluation. First, the source program was written in a normal style, referencing previously defined functions, and hence containing free variables. It is not necessary to abstract over such free variables before reification can occur. Second, several of the functions are polymorphic (e.g. \texttt{find}, \texttt{out}, \texttt{first} and \texttt{second}), and the implementation residualizes them without any explicit monomorphizing annotations. Third, the residual program contains an inductive structure, a term. The final extension, the residualization of a recursive function is not illustrated in this example.

12 Limitations

It is possible to force our type-directed, reification based, partial evaluator into an infinite loop. Consider the function \texttt{upto}:

\begin{verbatim}
val upto =
  fix upto => fn low => fn high =>
    if low > high then nil
    else cons(low, upto (low+1) high);
\end{verbatim}

It has type: \texttt{int -> int -> List int} and is a total function, it terminates on all integer input. If it is partially applied to an integer, say 1, a function of type: \texttt{int -> List int} is returned. Reification of this function causes an infinite loop.

The function returned by a \texttt{Y} combinator is "smart" but reconstructs itself only when applied to dynamic value. The value this function is applied to (1) is completely known, and in each recursive call it remains completely known. Termination of the function depends upon the dynamic parameter \texttt{high}, which is not known and the reification infinitely unfolds the fixed-point combinator.

This happens whenever a function defined with the fixed point combinator does not have its termination controlled by its first argument. This is a serious impediment and needs further study. Fortunately users can control this problem and write functions that avoid it if necessary.

As discussed by Danvy, type-directed partial evaluators may also cause duplication of code. This comes directly from splitting contexts over sum abstractions. As discussed by Danvy, this problem can be solved by residualizing local \texttt{let} expressions.

13 Related work

This work was inspired by the work of Danvy[9], which first demonstrated to the author the concept of reification based partial evaluation. Our use of a \texttt{value} type which embeds type information is a major contribution to Danvy's work. Danvy used the Scheme compiler as his reduction engine. This constraint did not allow him the flexibility needed for self describing values. Such values enable the extensions for free variables, and polymorphism, and we view this as one of the major contributions of this paper.

The use of an injection constructor (\texttt{vdyn}) which allows the embedding of the syntactic domain into the semantic domain has roots in earlier work in the use of catamorphisms as structured control operators[12], and in meta programming systems where code is a first class value[21].

The traditional\textsuperscript{3} partial evaluation literature describes two separate techniques which are used to control the complexity of performing symbolic evaluation of the source program given its static inputs. The first, offline partial evaluation\cite{18, 7}, uses an initial phase, called binding-time analysis, which uses only the fact that an input is static, to construct an annotated program. This annotated program is then residualized, executing the static components and rebuilding the dynamic components to construct the residual program.

On-line partial evaluators\cite{20, 19}, on the other hand, use the actual values associated with the static inputs to symbolically execute the source to build the residual program. We consider our partial evaluator on-line since the implementation of "smart" primitives actually probes the actual values of the static inputs.

Two of the harder problems in partial evaluation are pushing a static context over a dynamic branch such as an \texttt{if} or \texttt{case}, and handling higher order functions. The first has been handled by continuation based specialization\cite{6, 4}, and the second by a closure analysis\cite{14, 15, 3, 5}. In reification based systems, higher order functions are treated like any other function, and static contexts are handled implicitly by the use of the \texttt{shift} and \texttt{reset} control operators which abstract the current context and push it into the clauses of the \texttt{case}.

The delimited control operators \texttt{shift} and \texttt{reset}\cite{11} seem to be necessary to do reification over sums. Both the author and Danvy\textsuperscript{3} have experimented with \texttt{reify} and \texttt{reflect} operators with explicit continuation parameters and have found them to be problematic. The implementation for \texttt{shift} and \texttt{reset} in SML used in our implementation can be found in the literature\cite{13}.

Expansion of a value to reflect its type has been used to perform binding-time improvements\cite{10}. The technique of using expansion-reduction systems to reach normal forms is well known in the rewriting community, especially the use of \texttt{eta-expansion}\cite{17}.

Recent work has used such techniques to construct the inverse of the evaluation functional\cite{2} and to demonstrate that every term (in a combinator form) of system \texttt{F} has a normal form\cite{1}. In the latter work, a constructive proof is used to extract an ML program remarkably similar to the reification based partial evaluator for the polymorphic lambda calculus.

Recent work by John Hughes\cite{16} builds another framework for type based partial evaluation. Here, rather than base the propagation of static information on the unfolding of functions a type inference-like analysis is used instead. This technique has been quite effective in removing run-time datatype tags, when the specialized version of a program no longer needs them.

\textsuperscript{3}As opposed to type-directed.

\textsuperscript{4}Private Communication.
14 Conclusion

Partial evaluators can be constructed using a new paradigm: type-directed reification. This paradigm leads to systems that are simple to construct, small in size, and need no analysis other than type inference. We presented a partial evaluator which extends the work of Danvy which can handle: polymorphically typed functions, free variables in terms, an explicit fixed-point operator (or recursion), and inductive datatypes. It is based upon a novel technique that embeds type and other information in the implementation of the semantic domain, so that every value implicitly contains enough type information to reify itself. The implementation passes around type tags at partial-evaluation time but need not do so when all partial evaluation is concluded. We consider type-embedding values, and the ability to handle primitive functions in a systematic way, two important contributions of this paper.

Of course, we could have viewed partial evaluation as an expansion-reduction system performing normalization in the syntactic domain alone, using symbolic evaluation “under the lambda” to reach normal forms. The expansion-reduction properties become clearly evident in such a view. Expansion is necessary whenever \( \beta \)-reduction cannot proceed because an abstraction is applied to expression of the wrong shape.

The most important reason not to base an implementation on this view is that it requires complicated machinery to deal with environments, bound variables and substitutions. The beauty of our implementation strategy is that all these problems are encapsulated in the eval function and are handled in a completely standard and elegant manner and never need be considered again. In addition, by using two domains we make precise the distinction between values and terms. This distinction helped clarify for us many of the subtleties in Danvy's paper where both values and expressions are just s-expressions in Scheme.

One of the elegant features of implementing a partial evaluator in this fashion is that the symbolic reduction mechanism is the operational semantics. There is no question if the semantics and the behavior of the symbolic execution mechanism coincide.

References


A Appendix

This appendix contains the complete example referenced in Section 11. The code for the example appears in Figure 10. We define two datatypes which along with the List datatype of Section 9 are used to implement terms and substitutions. A term is either a variable, an integer constant or an infix operator. A term is encoded as the fixed-point of the sum type, i.e., \(\text{term} = (\text{Fix } x \Rightarrow T \, x)\). The \(\Sigma\) type encodes a maybe type, with either a single element or nothing. Substitutions are implemented as maybe lists of string cross terms: \(\Sigma(\text{List}(\text{string} \times \text{Fix } x \Rightarrow T \, x))\). nothing indicates the failure substitution.

The polymorphic function \(\text{find}\) searches a list of \((\text{string}\times\text{a})\) pairs for one whose first component is \(s\), and returns just\((a)\) if it is found, and nothing otherwise. Note that the definition of \(\text{find}\) uses the \(\text{rfun}\) (recursive-function) syntactic sugar. Using this notation \(\text{rfun find } s (\text{In } x) = \ldots\) is equivalent to \(\text{val find } = \text{fn } s \Rightarrow \text{fix find } \Rightarrow \text{fn } (\text{In } x) \Rightarrow \ldots\).

The function \(\text{term eq}\) tests two terms for equality, and \(\text{subst}\) applies a substitution to a term. Both \(\text{match}\) and \(\text{rewrite}\) were described earlier.

The residual program of Section 11 was constructed in the following manner. The program in Figure 10 was loaded into the system. A term representing \((x + y) + z \Rightarrow x + (y + z)\) was constructed. The curried function \(\text{rewrite}\) was applied to this term. A function with type \(\text{term} \rightarrow \text{term}\) was returned. This function was rafified producing the residual function displayed.

```
sum \(\Sigma\) a = \text{Var \, string}
   | \text{Op } (a \, \ast \, \text{string} \, \ast \, a)
   | \text{Int } (\text{int})

sum \(\Sigma\) a = \text{Nothing \, unit} | \text{Just } a;

val find = \text{fn } s \Rightarrow \text{fix } f \Rightarrow \text{fn } (\text{In } x) \Rightarrow
\text{case } x \text{ of }
| \Sigma \text{nil} \Rightarrow \text{Nothing }()
| \text{Cons}(a,\Sigma) \Rightarrow
   \begin{cases}
   \text{if streq}(a, s) \Rightarrow \text{Just } z \text{ else find } s\ b;
   \end{cases}
\text{fun} \text{out } (\text{In } x) = x;
\text{fun} \text{first } (x, y) = x;
\text{fun} \text{second } (x, y) = y;
\text{rfun} \text{term eq } (\text{In } t\ i) = (\text{In } x) =
   \begin{cases}
   \text{Var } s \Rightarrow (\text{case } x \text{ of } \text{Var } t \Rightarrow \text{streq}(s, t)
   \quad | \_ \Rightarrow \text{false})
   \end{cases}
| \text{Op } (m, s, a) \Rightarrow
   \begin{cases}
   \text{case } x \text{ of } \text{Op } (a, b, c) \Rightarrow
   \quad | \text{if streq}(a, b)
   \quad \quad | \text{then term eq } m \text{ a}
   \quad \quad \quad | \text{then term eq } n \ c \text{ else false} \quad \text{else false}
   \end{cases}
   \begin{cases}
   | \_ \Rightarrow \text{false})
   \end{cases}
| \text{Int } n \Rightarrow (\text{case } x \text{ of } \text{Int } m \Rightarrow n \text{ m}
   \quad | \_ \Rightarrow \text{false})
\end{cases}
\text{rfun subst sig } (\text{In } t) =
\text{let} \text{val} \ f = \text{subst sig in}
\begin{cases}
\text{case } t \text{ of }
   \text{Var } v \Rightarrow (\text{case } \text{find } v \text{ sig of}
   \quad | \text{Nothing} \Rightarrow (\text{In } (\text{Var } v))
   \quad \quad | \text{Just } w \Rightarrow w)
   \end{cases}
| \text{Op } (t1, s, t2) \Rightarrow \text{In } (\text{Op } (f \, t1, s, f \, t2))
| \text{Int } i \Rightarrow \text{In } (\text{Int } i)
\end{cases}
\end{cases}
\text{end};
\text{rfun} \text{match pat m sig} \text{ term} =
\begin{cases}
\text{case } (\text{msig} \text{ term}) \text{ of }
   \text{Nothing } () \Rightarrow \text{Nothing } ()
   \end{cases}
| \text{Just } (\Sigma) \Rightarrow
\begin{cases}
\text{case } (\text{out pat}) \text{ of }
   \text{Var } u \Rightarrow
   \begin{cases}
   \text{case } \text{find } u \text{ sig of }
   \quad | \text{Nothing} () \Rightarrow
   \quad \quad | \text{Just } (\text{cons}(u, \text{term}, \text{sigma}))
   \quad \quad \quad | \text{Just } w \Rightarrow \text{term if \text{term eq } w}
   \quad \quad \quad \quad | \text{then Just sigma}
   \quad \quad \quad \quad \quad | \text{else Nothing } ()
   \end{cases}
   \end{cases}
| \text{Op } (t1, s1, t12) \Rightarrow
   \begin{cases}
   \text{case } \text{term eq } \text{ of }
   \quad | \text{Op } (t21, s2, t22) \Rightarrow
   \quad \begin{cases}
   \text{if streq}(s2, (s1)) \Rightarrow
   \quad | \text{then match } t1 (\text{match } t12 \text{ msig} \text{ term}22) \text{ t21}
   \quad \quad | \text{else Nothing } ()
   \quad \quad \quad | \_ \Rightarrow \text{Nothing } ()
   \end{cases}
   \end{cases}
| \text{Int } u \Rightarrow
   \begin{cases}
   \text{case } \text{term eq } \text{ of }
   \quad | \text{Int } u \Rightarrow
   \quad \begin{cases}
   \text{if u } \Rightarrow \text{msig}
   \quad \quad | \text{then msig}
   \quad \quad \quad | \text{else Nothing } ()
   \quad \quad \quad \quad | \_ \Rightarrow \text{Nothing } ()
   \quad \quad \quad \quad \quad | \_ \Rightarrow \text{Nothing } ()
   \end{cases}
   \end{cases}
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\text{end};
\text{fun} \text{rewrite rule term} =
\text{let} \text{val} \ lhs = \text{first rule}
\text{val} \ rhs = \text{second rule}
\text{val} \ ms = \text{match } \text{lhs } (\text{Just } []) \text{ term}
\begin{cases}
\text{case } \text{ms} \text{ of }
   \text{Nothing } () \Rightarrow \text{term}
   | \text{Just } (\Sigma) \Rightarrow \text{subst sig} \text{ rhs}
\end{cases}
\end{cases}
\text{end};
```

Figure 10: Full Pattern Matching Code