The Partial Spechilada

Nikolaj S. Bjørner
bjorner@kestrel.edu
Kestrel Institute

1 Introduction

This paper examines different ways to represent and validate core program transformations. As the leading example, we take finite differencing, and subject it to alternative encodings. Our first encoding is to represent finite differencing equationally, and appeal to rewrite based simplification to apply it. The second encoding is to represent the transformation as a library refinement and use *diagram pushouts* to apply it. Finally we consider an encoding as a meta-program that works on the abstract syntax tree of Specware terms. While meta-programming allows finegrained control over the transformations it may be less declarative than object level axiomatization, and often less transparent. The assurrance problem for meta-programs becomes an issue when having to rely on a *dynamically growing set of transformations, in a fastly moving scenario.* While this can be recast as a compiler correctness problem, we shall discuss type based partial correctness criteria, an even more interestingly, frameworks for automatic inference of meta-typeability: the meta programs transform well typed programs into well typed programs.

The exposition is technical, although we only present elementary concepts and examples to ease the readability. The high points of this paper can be summarized as equation (4) and Figure 9.

2 Specware and Designware

Kestrel has for a number of years been developing and researching tools for program specification and synthesis. The Kestrel Institute Development System, KIDS, is a stable program transformation system. While highly effective at its design goals, it does not support data type refinement. The more recent tool Specware is based on concepts from category theory and addresses this point, while in itself does not deal with program transformation. On top of Specware we are developing program transformation capabilities. This extension is called Designware.

3 Transformations as Equations

Finite differencing, or strength reduction, is a program transformation that incrementally computes intermediate values in loops. A simple example is the function

$$\mathrm{sosq}(n : \mathsf{Nat}, m : \mathsf{Nat}) \quad = \quad \textbf{if} \;\; n \geq m \; \textbf{then} \;\; 0 \; \textbf{else} \;\; n^2 + \mathrm{sosq}(n+1, m) \quad (1)$$

that when supplied with arguments n and m computes the sum of squares from n to $m-1$. Since

$$(n+1)^2 = n^2 + 2 \cdot n + 1$$

it is possible to avoid computing $(n+1)^2$ by remembering the value of n^2 and then adding $2 \cdot n + 1$, which itself can be computed using a left shift and a bitwise or with 1. Thus, the program

$$\begin{aligned}
\mathrm{sosq}(n : \mathsf{Nat}, m : \mathsf{Nat}) \quad &= \quad \mathrm{sosq}'(n^2, n, m) && (2)\\
\mathrm{sosq}'(n_{sq}, n, m) \quad &= \quad \textbf{if} \;\; n \geq m \; \textbf{then} \;\; 0 \; \textbf{else} \\
& \qquad n_{sq} + \mathrm{sosq}'(n_{sq} + \mathit{lshift}(n) + 1, n+1, m) && (3)
\end{aligned}$$

avoids multiplication entirely in the main loop. We can also make sosq' tail-recursive in a further optimization based on the fact that + is commutative. Of course, this example is inherently artificial since sosq is computable by a closed term involving cubes. Nevertheless, not every loop is reducible in this way, and finite differencing has a distinguished place as a useful program transformation.

3.1 An equational characterization of finite differencing

The use of finite differencing is pervasive in program transformation systems such as KIDS, Hylo, RAPS, and Designware, among others.

In essence, finite differencing adds extra arguments to a recursive function that pre-calculate selected subterms. Finite differencing improves the program if the new arguments are less expensive to update in recursive calls than the old ones. We can apply finite differencing in three steps:

1. Select a subterm to be replaced by an induction variable.
2. Generate the finitely differenced version of the recursive function.
3. Simplify the resulting function using constraints on the induction variable.

Using higher-order functions, we can characterize steps 1 and 2 using a single equation:

$$\begin{array}{l}\forall X, Y\ .\ \mathbf{fix}(\lambda f\ a.\ X\ (Y\ a)\ f\ a) = \\ \qquad \lambda a.\mathbf{fix}(\lambda g\ (b,c).X\ c\ (\lambda\ d.\ g\ (d,\ Y\ d))\ b)\ (a,\ Y\ a)\end{array} . \tag{4}$$

With type annotations, this equation reads:

$$\begin{array}{l}\forall X : \gamma \to (\alpha \to \beta) \to \alpha \to \beta, \\ \forall Y : \alpha \to \gamma \\ \quad \mathbf{fix}\left(\begin{array}{c}\lambda f : \alpha \to \beta,\ a : \alpha. \\ X\ (Y\ a)\ f\ a\end{array}\right) \quad = \\ \quad \lambda a : \alpha.\mathbf{fix}\left(\begin{array}{c}\lambda g : \alpha \times \gamma \to \beta,\ (b,c) : \alpha \times \gamma. \\ X\ c\ (\lambda\ d : \alpha.\ g\ (d,\ Y\ d))\ b\end{array}\right)(a,\ Y\ a)\end{array} \tag{5}$$

where $\mathbf{fix}$ is the fixpoint operator of type $\forall\alpha\ .\ (\alpha \to \alpha) \to \alpha$ and satisfies $\mathbf{fix}\ f = f(\mathbf{fix}\ f)$.

In fact, we will use subtypes to strengthen the right hand side with the additional information that the λ-bound c satisfies the invariant $c = (Y\ b)$. Thus, the right hand side takes the form

$$\begin{array}{r}\lambda a : \alpha.\mathbf{fix}\left(\begin{array}{c}\lambda g : \delta \to \beta,\ (b,c) : \delta. \\ X\ c\ (\lambda\ d : \alpha.\ g\ (d,\ Y\ d))\ b\end{array}\right)(a,\ Y\ a) \\ \text{where} \quad \delta = \{(b,c) \mid Y\ b = c\}\end{array}$$

Notice, however, that we have left the subtype coercions implicit, as these can be inferred and checked automatically.

3.2 Calculating sosq'

Returning to the sum of squares example, we present the original program in a form suitable for applying equation (5):

$$\text{sosq} \quad = \quad \mathbf{fix}\left(\begin{array}{c}\lambda\text{sosq} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat},\ (n,m) : \mathsf{Nat} \times \mathsf{Nat}. \\ X\ (Y\ a)\ f\ a\end{array}\right) \tag{6}$$

$$X \quad = \quad \lambda n_{sq}\ .\ \lambda\text{sosq}\ .\ \lambda(n,m)\ .\ \left(\begin{array}{c}\mathbf{if}\ \ n \geq m\ \mathbf{then}\ \ 0\ \mathbf{else} \\ n_{sq} + \text{sosq}(n+1,m)\end{array}\right) \tag{7}$$

$$Y \quad = \quad \lambda(n,m)\ .\ n^2 \tag{8}$$

After rewriting using equation (5), we have

$$\lambda(n,m) : \mathsf{Nat} \times \mathsf{Nat}\ .\ \text{sosq}'\ ((n,m),\ n^2)$$

where

$$\text{sosq}' \quad = \quad \textbf{fix} \left(\begin{array}{l} \lambda \text{sosq}', ((n,m),\ n_{sq}). \\ \qquad\qquad X\ n_{sq}\ (\lambda d\ .\ \text{sosq}'\ (d,\ Y\ d))\ (n,m) \end{array} \right) \tag{9}$$

Simplifying the inner expression, we obtain precisely the version of sosq' from equation (3):

$$\begin{array}{ll} & X\ n_{sq}\ (\lambda d\ .\ \text{sosq}'\ (d,\ Y\ d))\ (n,m) \qquad (10) \\ = & \quad \{\text{by } \beta \text{ expansion}\} \\ & \textbf{if}\ \ n \geq m\ \textbf{then}\ \ 0\ \textbf{else}\ n_{sq} + \text{sosq}'\ ((n+1,m),\ (n+1)^2) \qquad (11) \\ = & \quad \{\text{by the rewrites } (x+1)y = xy + y, x(y+1) = xy + x\} \\ & \textbf{if}\ \ n \geq m\ \textbf{then}\ \ 0\ \textbf{else}\ n_{sq} + \text{sosq}'\ ((n+1,m),\ (n^2 + 2 \cdot n + 1)) \ (12) \\ = & \quad \{\text{by applying the subsort property on } n_{sq}\} \\ & \textbf{if}\ \ n \geq m\ \textbf{then}\ \ 0\ \textbf{else}\ n_{sq} + \text{sosq}'\ ((n+1,m),\ (n_{sq} + 2 \cdot n + 1))(13) \end{array}$$

Specware has a rewrite-based simplification engine that uses higher-order matching. When simplifying terms in context, subtype information, such as $n_{sq} : \{x : \mathsf{Nat} \mid n^2 = x\}$, is used to supply the rewrite engine with auxiliary rewrites (in our example $n^2 = n_{sq}$). Thus, when given the equalities (7) and (8), and the hint that $(n+1)^2 = n^2+2n+1$, the rewrite engine rewrites (10) into (13). This is illustrated later in Figure 7.

3.3 Applying finite differencing

Similarly, the higher-order rewrite engine can perform finite differencing by matching a term against the left hand side of equality (5) and replacing it with the right hand side of (5). In this connection, X and Y act as meta variables that can be bound to arbitrary closed subterms, as in equations (7) and (8).

Unfortunately, a single application of higher-order matching may return several matching substitutions. Thus, we need to consider how to choose which substitution to use. Of course, in our example, there is only one other substitution, and it is not very interesting: $Y' = \lambda(n,m)\ .\ n$ and X' is X except that we replace n_{sq} by n_{sq}^2.

To show the correctness of the equational presentation of finite differencing, we simply need to verify equation (5). The proof uses fixpoint induction and the monotonicity properties of $\sqsubseteq$. In this context, fixpoint induction is the rule:

$$\frac{\begin{array}{l} M\ \bot \sqsubseteq N \\ \forall Z\ .\ M\ Z \sqsubseteq N \ \ \Rightarrow \ \ M\ (M\ Z) \sqsubseteq N \end{array}}{\textbf{fix}\ M \sqsubseteq N}$$

For example:

$$
\begin{array}{ll}
& (\lambda f\ a.\ X\ (Y\ a)\ f\ a)\ \bot \\
= & \qquad \{\text{by } \beta \text{ reduction}\} \\
& \lambda a.\ X\ (Y\ a)\ \bot\ a \\
\sqsubseteq & \qquad \{\text{since } \bot \sqsubseteq X \text{ for every } X\} \\
& \lambda a.\ X\ (Y\ a)\ (\lambda\ d.\ \mathsf{G}\ (d,\ Y\ d))\ a \\
= & \qquad \{\text{by folding with the definition of } \mathbf{fix}\} \\
& \lambda a.\ \underbrace{\mathbf{fix}(\lambda g\ (b,\ c).X\ c\ (\lambda\ d.\ g\ (d,\ Y\ d))\ b)}_{\mathsf{G}}\ (a,\ Y\ a)
\end{array}
$$

$$
\begin{array}{ll}
& (\lambda f\ a.\ X\ (Y\ a)\ f\ a)\ ((\lambda f\ a.\ X\ (Y\ a)\ f\ a)\ Z) \\
= & \qquad \{\text{by } \beta \text{ contraction}\} \\
& \lambda a.X\ (Y\ a)\ ((\lambda f\ a.\ X\ (Y\ a)\ f\ a)\ Z)a \\
\sqsubseteq & \qquad \{\text{by the induction hypothesis}\} \\
& \lambda a.X\ (Y\ a)\ (\lambda a\ .\mathsf{G}(a, Y\ a))a \\
= & \qquad \{\text{by abstracting } a \text{ and } Y\ a\} \\
& \lambda a.(\lambda (b,c)\ .\ X\ c\ (\lambda a\ .\mathsf{G}(a, Y\ a))b)(a, Y\ a) \\
= & \qquad \{\text{by folding the definition of } \mathsf{G}\} \\
& \lambda a.\ \underbrace{\mathbf{fix}(\lambda g\ (b,\ c).X\ c\ (\lambda\ d.\ g\ (d,\ Y\ d))\ b)}_{\mathsf{G}}\ (a,\ Y\ a)
\end{array}
$$

4 Transformations as pushouts

In this section, we describe Designware's framework for program transformation. Designware is a system developed at Kestrel that uses category theory as a tool for organizing program development. In Designware, transformations on specifications (and, in particular, programs) are realized using *pushouts*. Naturally, we use finite differencing as an example to illustrate pushouts.

4.1 Specifications

Consider the two specifications **FD-Source** and **FD-Target** below in Figure 1 and 2.

4.2 Morphisms

We can relate **FD-Source** and **FD-Target** using a *morphism*. For this discussion, we define a morphism as a mapping from sort symbols in the source to sort terms in the target and from op symbols in the source to (closed) op terms in

spec FD-Source =
  sort α, β, γ
  op $X : \gamma \to (\alpha \to \beta) \to \alpha \to \beta$
  op $Y : \alpha \to \gamma$
  def $f(a) = X\ (Y\ a) f a$
end-spec

Figure 1: Finite differencing source

spec FD-Target =
  sort α, β, γ
  op $X : \gamma \to (\alpha \to \beta) \to \alpha \to \beta$
  op $Y : \alpha \to \gamma$
  def $f'(b, c) = X\ c\ (\lambda\ d.\ f'(d, Y\ d))\ b$
  def $f(a) = f'(a, Y\ a)$
end-spec

Figure 2: Finite differencing target

the target. Thus, a morphism shows how to translate sorts and operations from one specification to another. Furthermore, the resulting translation is required to map axioms in the source to theorems in the target. Morphisms are also called *refinements*; a morphism from theory A to theory B shows how B is a refinement of A.

In our example, we can relate **FD-Source** to **FD-Target** using the morphism

$$\alpha \mapsto \alpha,\ \beta \mapsto \beta,\ \gamma \mapsto \gamma,\ X \mapsto X,\ Y \mapsto Y,\ f \mapsto f$$

The source specification contains one axiom, namely the equality $f(a) = X\ (Y\ a)\ f\ a$. This equality is provable using the definitions in the target, as we saw in Section 3.3.

This morphism can be stored in a library and (re)used to apply finite differencing. In our example, we begin with this specification:

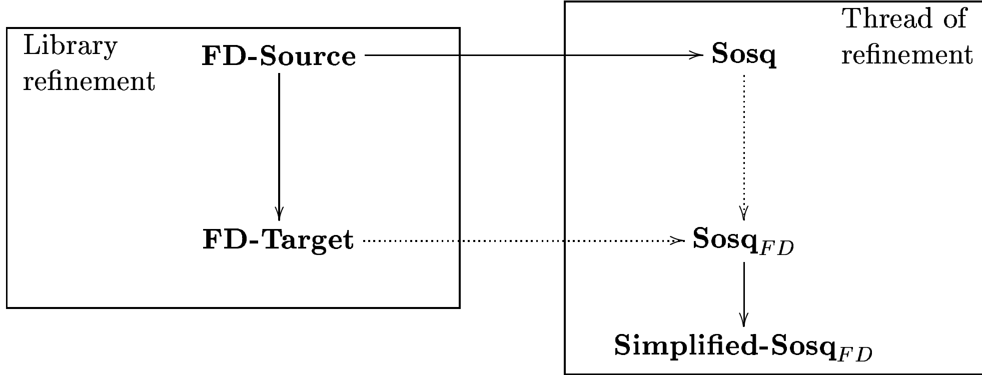
spec Sosq =
  def $\mathrm{sosq}(n, m) =$ **if** $n \geq m$ **then** 0 **else** $n^2 + \mathrm{sosq}(n+1, m)$
end-spec

Figure 3: Sum of squares

Then, using higher-order matching, we construct a morphism from **FD-Source** to **Sosq**. The morphism we are interested in is not surprisingly characterized by equations (7) and (8).

4.3 Pushouts

Given the morphism from **FD-Source** to **FD-Target** and the morphism from **FD-Source** to **Sosq**, we perform a *pushout* to compute $\mathbf{Sosq}_{FD}$, the finitely differenced sum of squares:



The resulting specification $\mathbf{Sosq}_{FD}$ contains equation (3). The pushout operation computes $\mathbf{Sosq}_{FD}$ and the two dashed morphisms into it automatically. In fact, $\mathbf{Sosq}_{FD}$ is the least constrained specification that admits morphisms from **FD-Target** and **Sosq** and that makes the above square commute.

After applying finite differencing, we can apply additional simplification steps to construct a simplified specification **Simplified-Sosq**$_{FD}$ and a morphism from $\mathbf{Sosq}_{FD}$ to it.

spec Simplified-Sosq-FD =
  def $\text{sosq}(n, m) = \text{sosq}'((n, m), n^2)$
  def $\text{sosq}'((n, m), n_{sq}) =$ **if** $n \geq m$ **then** 0 **else**
    $n_{sq} + \text{sosq}'((n + 1, m), n_{sq} + 2 \times n + 1)$
end-spec

Figure 4: Finitely differenced, simplified, Sum of squares

4.4 Summary

This section presented the basic concepts of the Designware framework: specifications, morphisms, and pushouts. Designware uses morphisms to represent both program transformations and data type refinements. To verify a transformation, we check that the morphism representing it indeed sends axioms to theorems. To apply a transformation, we first find a morphism from the source of the transformation morphism to the specification to be transformed using higher-order matching or witness finding. Then we compute the pushout of this morphism and the transformation morphism. The pushout operation can be implemented efficiently using the standard algorithm for union-find.

5 Transformations as Meta Programs

So far, we have examined two approaches to applying program transformations. The first approach applied transformations using a rewriter based on higher-order matching. The second approach applied transformations by pushout in the category of specifications and morphisms. Both styles required higher-order matching to determine where to apply the transformation.

Common transformations may be encoded more conveniently using meta programs, that is, as programs that manipulate other programs. For example, in the actual Designware implementation, finite differencing is performed by a specialized tactic. The user selects a subterm occurring in some definition, and the tactic creates a new definition with the subterm added as an extra argument. This process does not require any higher-order matching or pushout computation. Instead, the finite differencing transformation is hardwired as a meta-program.

The effect of applying the finite differencing tactic, followed by simplification is illustrated in Figures 5, 6, and 7.

```
Specware Interaction
File Edit Transformation Type Check Settings Specifications          Help Cancel
spec Sosq =
 op sosq : Nat * Nat -> Nat
 def sosq (n, m) = if n >= m then 0 else (n * n) + sosq(n + 1, m)
 axiom n-times-n = fa(n : Nat) ((n + 1) * (n + 1)) = (((n * n) + (2 * n)) + 1)
end-spec

--**-XEmacs: *Specware*      (SPECWARE Font)----All----------------------------------
```

Figure 5: Original sum of squares specification.

```
Specware Interaction
File Edit Transformation Type Check Settings Specifications          Help Cancel
spec Sosq-1 =
 op sosq : Nat * Nat -> Nat
 def sosq (n, m) = sosq0((n, m), n * n)
 op sosq0 : {((n, m), fdV) : (Nat * Nat) * Nat | (n * n) = fdV} -> Nat
 def sosq0 ((n, m), fdV) =
     if n >= m
       then 0
     else fdV + (case (n + 1, m)
                   of (n, m) -> sosq0((n, m), n * n))
 axiom n-times-n = fa(n : Nat) ((n + 1) * (n + 1)) = (((n * n) + (2 * n)) + 1)
end-spec
--**-XEmacs: *Specware*      (SPECWARE Font)----All----------------------------------
```

Figure 6: Finitely differenced sum of squares specification.

Before presenting the meta program, let us re-examine **FD-Target**:

$$\textbf{def } f'(b, c) = X\ c\ (\lambda\ d.\ f'(d, Y\ d))\ b$$

```
Specware Interaction
File  Edit  Transformation  Type Check  Settings  Specifications                Help  Cancel
spec Sosq-2 =
 op sosq : Nat * Nat -> Nat
 def sosq (n, m) = sosq0((n, m), n * n)
 op sosq0 : {((n, m), fdV) : (Nat * Nat) * Nat | (n * n) = fdV} -> Nat
 def sosq0 ((n, m), fdV) =
     if n >= m then 0 else fdV + sosq0((n + 1, m), (fdV + (2 * n)) + 1)
 axiom n-times-n = fa(n : Nat) ((n + 1) * (n + 1)) = (((n * n) + (2 * n)) + 1)
end-spec
--**-XEmacs: *Specware*      (SPECWARE Font)----All----------------------------------
Fontifying *Specware*... done.
```

Figure 7: Simplified sum of squares specification.

def $f(a) = f'(a, Y\ a)$

In this context we can rewrite the definition of f', first by α-renaming, then by folding with the definition of f.:

$$\begin{aligned} f'(a,c) &= X\ c\ (\lambda\ a.\ f'(a, Y\ a))\ a && (14)\\ &= X\ (Y\ a)(\lambda\ a.\ f'(a, Y\ a))\ a && (15)\\ &= X\ (Y\ a)\ f\ a && (16) \end{aligned}$$

Apparently we have obtained nothing, as this is almost the original definition of f, without the induction variable c. On the other hand, this equation suggests a simple way to implement finite differencing: introduce an auxiliary function f' with an argument c that satisfies $Ya = c$, make the body of f' be the body of f and let f call f' as done in **FD-Target**. In the resulting specification, replace every use of f by $\lambda a.\ f'(a, Y\ a)$ (this corresponds to unfolding f by its immediate definition). We can also simplify the body of f' using the rewrite $Y\ a = c$.

This yields Specware's actual implementation of finite differencing. Specifically, suppose that we have selected a subterm N in a definition $f(a) = M$ of a specification *spc*. We pass f, a, N, M and *spc* as arguments to the following finite differencing transformation.

$$\begin{array}{l} \textbf{def}\ \mathrm{fd}(f, a, N, M, spc) = \\ \quad \textbf{let} \\ \qquad f' = \mathrm{fresh}() \\ \qquad b = \mathrm{fresh}() \\ \qquad spc = spc \dagger [f \mapsto \lceil \lambda a. f'(a, N) \rceil] \dagger [f' \mapsto \lceil \lambda(a, b).M \rceil] \\ \qquad unfold = \lambda x\ .\ \textbf{if}\ \ x = f\ \textbf{then}\ \lceil \lambda a.\ f'(a, N) \rceil\ \textbf{else}\ \ x \\ \quad \textbf{in} \\ \qquad mapSpec\ unfold\ spc \end{array}$$

Figure 8: A meta program for finite differencing

In this tactic, we use Quine brackets to distinguish object from meta-level terms. The auxiliary function *mapSpec* maps the function *unfold* across every subterm occurring in the specification, and the map update function † overrides *spc* with a new definition.

5.1 Type correctness

Ideally, we would like to derive the above meta program for finite differencing automatically from equation 5. However, for the moment, we are willing to settle for an automatic proof that the above meta program does not introduce type errors into the programs it transforms. It turns out that we can provide this guarantee by type checking the meta program in an expressive, two-level type system.

Ordinarily, meta programs represent all object terms, independent of their object type, as values of a single meta type Term. That is, a term of object type Nat is represented as a value of meta type Term, and similarly for a term of object type String. The obvious solution is to split the single meta type Term into a family of meta types $\mathsf{Term}[s]$ indexed by their object level types. Thus, an object term of type Nat is represented as a value of meta type Term[Nat]. We also introduce types $\mathsf{Var}[s]$ and *opSorts* to represent variables and operations.

Using this type system, which can be embedded into the system LF, we can type the finite differencing meta program as shown in Figure 9.

$$
\begin{array}{l}
\textbf{def}\ \mathrm{fd}(f : \mathsf{Op}[\alpha \to \beta], a : \mathsf{Var}[\alpha], N : \mathsf{Term}[\gamma], M : \mathsf{Term}[\beta], spc) = \\
\quad \textbf{let} \\
\qquad f' : \mathsf{Op}[\{(a, b) : \alpha \times \gamma \mid N = b\} \to \beta] = \mathrm{fresh}() \\
\qquad b : \mathsf{Var}[\gamma] = \mathrm{fresh}() \\
\qquad spc = spc \dagger [f \mapsto \ulcorner \lambda a.f'(a, N) \urcorner] \dagger [f' \mapsto \ulcorner \lambda (a, b).M \urcorner] \\
\qquad unfold : \forall \alpha\ .\ \mathsf{Term}[\alpha] \to \mathsf{Term}[\alpha] \\
\qquad\qquad\qquad = \lambda x\ .\ \textbf{if}\ \ x = f\ \textbf{then}\ \ulcorner \lambda a.f'(a, N) \urcorner\ \textbf{else}\ \ x \\
\quad \textbf{in} \\
\qquad mapSpec\ unfold\ spc
\end{array}
$$

Figure 9: A type annotated meta program for finite differencing

Let's examine the typing of the *map* operator. *Map f t* applies f to each subterm of t. If object terms are formed using the constructors App, Lam, Var and Op, we can define *map* as:

$$
\begin{array}{lcl}
map\ f\ (\mathsf{App}\ M\, N) & = & f(\mathsf{App}\ (map\ f\ M)\, (map\ f\ N)) \\
map\ f\ (\mathsf{Lam}\ v\, .\, M) & = & f(\mathsf{Lam}\ v\, .\, map\ f\ M) \\
map\ f\ (\mathsf{Var}\, v) & = & f(\mathsf{Var}\, v) \\
map\ f\ (\mathsf{Op}\, g) & = & f(\mathsf{Op}\, g)
\end{array}
$$

Map then has the interesting type:

$$(\forall\alpha\ .\ \mathsf{Term}[\alpha] \to \mathsf{Term}[\alpha]) \quad \to \quad (\forall\beta\ .\ \mathsf{Term}[\beta] \to \mathsf{Term}[\beta])\ .$$

This type is not expressible in the standard Hindley-Milner polymorphic type system since the scope of the type variables α and β is not the whole term. However, this type *is* expressible in the system of rank-2 bounded polymorphism. Type inference for rank-2 bounded polymorphism is decidable *until* we allow recursive functions, such as *map*, at which point it becomes undecidable. On the other hand, this example applies rank-2 polymorphism to an interesting problem; it isn't contrived or pathological in the least. For further technical details, see [1].

In this connection, a noteworthy approach that allows more declarative style meta-programming by adding higher-order matching is described in [2].

5.2 Bound and free variables

A fundamental limitation of the above approach is that the typing $M : \mathsf{Term}[s]$ tells us nothing about the free variables of M. For example, the finite differencing transformation only makes sense if the only free variable in N is a. The above type system still allows us to type check meta-programs that expose bound variables and capture free variables.

Fortunately, this problem is solved in [3], which presents a calculus that includes automatic α-renaming of bound variables. The calculus forces α-renaming into the language implementation but in return provides a type system for checking that programs do not expose bound variables or capture free variables. Our example can with a few modifications be fed to this system and thus certified to be safe with respect to the standard variable scoping rules. The challenge remains to develop a type system that handles variable scoping without building in α-renaming.

6 Conclusion

We used the finite-differencing program to expose different ways to encode program transformation tactics. While the framework of Designware is general enough to accomodate tactics such as finite differencing as a declaratively given library refinement, we, at least in practice, are not making use of this, but instead encode it using a hardwired tactic written as a meta-program. The validation problem for meta-programs involves reasoning at two levels, which is clearly more complicated than proving theorems that only deal with one level. Luckily, unguided support for, partial correctness, such as meta-type safety, can be addressed using rank-2 polymorphism.

Acknowledgements I would like to thank the workshop organizers for giving me an opportunity to summarize thoughts otherwise not possible in my daily

work. I would also like to thank my colleagues at Kestrel Institute for a joy-full environment. Without David Espinosa, we would not have had the most elegant category theory-based bootstrapped synthesis system (this is a highly competitive field). He translated this paper from Danish to English. Many thanks also to Cordell Green for his subtle humor and to Dusko Pavlovic for his equally unsubtle personality. Doug Smith's steady, consistent, and always energetic drive stands out as the most stimulating source for new, interesting and challenging, projects. John Anton is always around to give his support and do the heavy lifting. As usual, this note had not been written without a good dosis of Richard Waldinger's favorite Dark Roasted French Sumatra coffee. His 24 hour feedback service on using SNARK in connection with Specware is simply amazing.

References

[1] N. Bjørner. Type checking meta-programs. In *LFM'99: Proceedings of Workshop on Logical Frameworks and Meta-languages*, 1999.

[2] Z. Hu and M. Takeichi. Calculation carrying programs. Technical Report METR 99-07, University of Tokyo, September 1999.

[3] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000*, volume ? of *Lecture Notes in Computer Science*, pages ?–? Springer-Verlag, Heidelberg, 2000.