

# Building a Supercompiler

## Abstract

We present a supercompilation algorithm for System  $F_c$  and we prove that the algorithm both terminates and preserves termination properties.

**Categories and Subject Descriptors** D.3.4 [Programming Languages]: Processors – Compilers, Optimization; D.3.2 [Programming Languages]: Language Classifications – Applicative (functional) languages

**General Terms** Languages, Theory

**Keywords** supercompilation, deforestation, System F

## 1. Introduction

We explore one part of the design space for a positive supercompiler. We deliberately sacrifice transformational power for reduced compilation time. Supero (?) occupies a different part of the design space: the resulting binaries are much faster.

Components: Positive Supercompilation using Leuschel’s extended homeomorphic embedding with Supero’s bowtie for generalisation. We improve the actual performance of the supercompiler by using a zipped term representation which allows us to quickly discard terms for testing.

Our contributions are:

- A faster algorithm for positive supercompilation using a zipper to represent the terms.
- A new generalisation and whistle operating on this zipped representation.
- Heuristics to prune terms early from testing.

## 2. Language

Our language of study is System  $F_c$  (?) extended with primitive values and explicit function names.

We let constructor symbols be denoted by  $K$ . Let  $F$  range over a set  $\mathcal{F}$  of global function definitions, where all functions have a specific arity.

The language contains integer values  $n$  and arithmetic operations  $\oplus$ , although these meta-variables can preferably be understood as ranging over primitive values in general and arbitrary operations on these. We let  $+$  denote the semantic meaning of  $\oplus$ .

A program is an expression with no free variables and all function names defined in  $\mathcal{F}$ . The intended operational semantics is

The outer context:

$$\mathcal{R} ::= \epsilon \mid \mathcal{G} : \mathcal{R}$$

Each frame:

$$\mathcal{G} ::= \square e \mid \square \tau \mid \square \blacktriangleright \gamma \mid \square \oplus e \mid e' \oplus \square \mid \text{case } \square \text{ of } \{p_i \rightarrow e_i\}$$

$$\rho ::= \epsilon \mid \nu : \rho$$

$$\nu ::= (h, \bar{x}, F, \mathcal{R})$$

$$\sigma ::= (\nu, e')$$

**Figure 1.** The definitions for Simon’s algorithm

$$\begin{aligned} \mathcal{D}_{\text{let}}[[e]]_{\mathcal{F}, \rho, L} \mathcal{R} \sigma &= \text{let } (e', \sigma') = \mathcal{D}[[e]]_{\mathcal{F}, \rho} \mathcal{R} \sigma \\ &\quad (\sigma_1, \sigma_2) = \text{storeSplit}(\sigma', x) \\ &\quad \text{in } (L \sigma_1 \text{ in } e', \sigma_2) \end{aligned}$$

**Figure 3.** Driving of let-statements

call-by-need. Capture free substitution expressions  $\bar{e}$  for variables  $\bar{x}$  in  $e'$  is denoted by  $[\bar{e}/\bar{x}]e'$ .

## 3. Supercompilation

Our supercompiler is defined as two mutually recursive functions that pattern-match on expressions and contexts in order to rewrite expressions. The first algorithm is called the *driving* algorithm, used to perform evaluation steps, and the second algorithm is called the *building* algorithm, which reassembles the transformed expression. These algorithms are defined in Figure 2. Two additional parameters appear as subscripts to the rewrite rules: a memoization list  $\rho$  and the set of global functions  $\mathcal{F}$ . The memoization list holds information about expressions already traversed and is explained more in detail in Section 3.1.

We use  $e'$  to denote  $\text{OutExprs}$ .

A reduction context  $\mathcal{R}$ , as defined in Figure 1, is a term containing a single hole  $[\ ]$ , which indicates the next expression to be reduced. The expression  $\mathcal{R}(e)$  is the term obtained by replacing the hole in  $\mathcal{R}$  with  $e$ .  $\overline{\square e}$  denotes a list of application contexts.

There is an ordering between rules; i.e., all rules must be tried in the order they appear. Rules R17-R21 are the default focusing rules that extend the given driving context  $\mathcal{R}$  and zoom in on the next expression to be transformed. If no other rule matches, such as for a single variable, rule R22 will match and call build to rebuild the expression.

The program is turned “inside-out” by moving the surrounding context  $\mathcal{R}$  into all branches of the case-statement through rules R28 and R29. Rule R15 has a similar mechanism for let-statements.

Definition of doBeta:

```
doBeta :: ScpEnv -> CoreExpr -> [CoreExpr] -> CoreExpr
doBeta env (Lam b body) args@(a:as)
  | isTypeArg a = doBeta env (substExpr (extendSubst emptySu
```

[Copyright notice will appear here once ‘preprint’ option is removed.]

General form:

$$\mathcal{D}[[e]]_{\mathcal{F},\rho} \mathcal{R} \sigma = (e', \sigma')$$

Evaluation Rules

$$\mathcal{D}[[e \blacktriangleright \gamma_1]]_{\mathcal{F},\rho} (\square \blacktriangleright \gamma_2 : \mathcal{R}) \sigma = \mathcal{D}[[e]]_{\mathcal{F},\rho} (\square \blacktriangleright (\gamma_1 \circ \gamma_2) : \mathcal{R}) \sigma \quad (\text{R1})$$

$$\mathcal{D}[[n_j]]_{\mathcal{F},\rho} (\text{case } \square \text{ of } \{n_i \rightarrow e_i\} : \mathcal{R}) \sigma = \mathcal{D}[[e_j]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R2})$$

$$\mathcal{D}[[n]]_{\mathcal{F},\rho} (n_1 \oplus \square : \mathcal{R}) \sigma = \mathcal{D}[[n_2]]_{\mathcal{F},\rho} \mathcal{R} \sigma, \text{ where } n_2 = n_1 + n \quad (\text{R3})$$

$$\mathcal{D}[[F]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \mathcal{D}_{app}(F)_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R4})$$

$$\mathcal{D}[[K_j]]_{\mathcal{F},\rho} (\overline{\square e} : \text{case } \square \text{ of } \{K_i \bar{x}_i \rightarrow e_i\} : \mathcal{R}) \sigma = \mathcal{D}[[\text{let } \bar{x}_j = \bar{e} \text{ in } e_j]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R5})$$

$$\mathcal{D}[[K_j]]_{\mathcal{F},\rho} (\overline{\square e} : \square \blacktriangleright \gamma : \text{case } \square \text{ of } \{K_i \bar{x}_i \rightarrow e_i\} : \mathcal{R}) \sigma = \mathcal{D}[[\text{let } \bar{x}_j = \bar{e} \text{ in } e_j]]_{\mathcal{F},\rho} \mathcal{R} \sigma, \text{ XXX: KPush} \quad (\text{R6})$$

$$\mathcal{D}[[\lambda x. e]]_{\mathcal{F},\rho} (\square e' : \mathcal{R}) \sigma = \mathcal{D}[[\text{let } x = e' \text{ in } e]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R7})$$

$$\mathcal{D}[[\lambda x. e]]_{\mathcal{F},\rho} (\square \blacktriangleright \gamma : \mathcal{R}) \sigma = \mathcal{D}[[\text{evalPush } \lambda x. e \blacktriangleright \gamma]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R8})$$

$$\mathcal{D}[[\lambda x. e]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \text{let } (e', \sigma_1) = \mathcal{D}[[e]]_{\mathcal{F},\rho} \epsilon \sigma \text{ in } \mathcal{B}[[\lambda x. e']]_{\mathcal{F},\rho} \mathcal{R} \sigma_1 \quad (\text{R9})$$

$$\mathcal{D}[[\Lambda a. e]]_{\mathcal{F},\rho} (\square \varphi : \mathcal{R}) \sigma = \mathcal{D}[[[\varphi/a]e]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R10})$$

$$\mathcal{D}[[\Lambda a. e]]_{\mathcal{F},\rho} (\square \blacktriangleright \gamma : \mathcal{R}) \sigma = \mathcal{D}[[\text{evalPush } \Lambda a. e \blacktriangleright \gamma]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R11})$$

$$\mathcal{D}[[\Lambda a. e]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \text{let } (e', \sigma_1) = \mathcal{D}[[e]]_{\mathcal{F},\rho} \epsilon \sigma \text{ in } \mathcal{B}[[\Lambda a. e']]_{\mathcal{F},\rho} \mathcal{R} \sigma_1 \quad (\text{R12})$$

$$\mathcal{D}[[\text{let } x = v \text{ in } e]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \mathcal{D}_{let}[[e]]_{\mathcal{F}',\rho}, \text{let } \mathcal{R} \sigma, \text{ where } \mathcal{F}' = \mathcal{F} \cup (F, v) \quad (\text{R13})$$

$$\mathcal{D}[[\text{let } x = y \text{ in } e]]_{\mathcal{F},\rho} \mathcal{R} \sigma \mid y \notin \text{SPLITVARS} = \mathcal{D}[[[y/x]e]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R14})$$

$$\mathcal{D}[[\text{let } x = e \text{ in } e']]_{\mathcal{F},\rho} \mathcal{R} \sigma \mid \text{linear } x \text{ e}' = \mathcal{D}[[[e/x]f]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R15})$$

$$\begin{aligned} & \text{otherwise} \\ & = \text{let } (e_1, \sigma_1) = \mathcal{D}[[e]]_{\mathcal{F},\rho} \epsilon \sigma \\ & \quad (e_2, \sigma_2) = \mathcal{D}[[e']]_{\mathcal{F},\rho} \mathcal{R} \sigma_1 \\ & \text{in } (\text{let } x = e_1 \text{ in } e_2, \sigma_2) \end{aligned}$$

$$\mathcal{D}[[\text{letrec } F = e \text{ in } e']]_{\mathcal{F},\rho} \mathcal{R} \sigma = \mathcal{D}_{let}[[e']]_{\mathcal{F}',\rho}, \text{letrec } \mathcal{R} \sigma, \text{ where } \mathcal{F}' = \mathcal{F} \cup (F, e) \quad (\text{R16})$$

Focusing Rules

$$\mathcal{D}[[e_1 \oplus e_2]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \mathcal{D}[[e_1]]_{\mathcal{F},\rho} (\square \oplus e_2 : \mathcal{R}) \sigma \quad (\text{R17})$$

$$\mathcal{D}[[e_1 e_2]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \mathcal{D}[[e_1]]_{\mathcal{F},\rho} (\square e_2 : \mathcal{R}) \sigma \quad (\text{R18})$$

$$\mathcal{D}[[e \varphi]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \mathcal{D}[[e_1]]_{\mathcal{F},\rho} (\square \varphi : \mathcal{R}) \sigma \quad (\text{R19})$$

$$\mathcal{D}[[\text{case } e \text{ of } \{p_i \rightarrow e_i\}]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \mathcal{D}[[e]]_{\mathcal{F},\rho} (\text{case } \square \text{ of } \{p_i \rightarrow e_i\} : \mathcal{R}) \sigma \quad (\text{R20})$$

$$\mathcal{D}[[e \blacktriangleright \gamma]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \mathcal{D}[[e]]_{\mathcal{F},\rho} (\square \blacktriangleright \gamma : \mathcal{R}) \sigma \quad (\text{R21})$$

Fallthrough

$$\mathcal{D}[[e]]_{\mathcal{F},\rho} \mathcal{R} \sigma = \mathcal{B}[[e]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R22})$$

Rebuilding Expressions

$$\mathcal{B}[[e']]_{\mathcal{F},\rho} (\square \oplus e_2 : \mathcal{R}) \sigma = \mathcal{D}[[e_2]]_{\mathcal{F},\rho} (e' \oplus \square : \mathcal{R}) \sigma \quad (\text{R23})$$

$$\mathcal{B}[[e']]_{\mathcal{F},\rho} (e_1 \oplus \square : \mathcal{R}) \sigma = \mathcal{B}[[e_1 \oplus e']]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R24})$$

$$\mathcal{B}[[e']]_{\mathcal{F},\rho} (\square e : \mathcal{R}) \sigma = \text{let } (e'', \sigma_1) = \mathcal{D}[[e]]_{\mathcal{F},\rho} \epsilon \sigma \text{ in } \mathcal{B}[[e' e'']]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R25})$$

$$\mathcal{B}[[e']]_{\mathcal{F},\rho} (\square \varphi : \mathcal{R}) \sigma = \mathcal{B}[[e' \varphi]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R26})$$

$$\mathcal{B}[[e']]_{\mathcal{F},\rho} (\square \blacktriangleright \gamma : \mathcal{R}) \sigma = \mathcal{B}[[e \blacktriangleright \gamma]]_{\mathcal{F},\rho} \mathcal{R} \sigma \quad (\text{R27})$$

$$\mathcal{B}[[x']]_{\mathcal{F},\rho} (\text{case } \square \text{ of } \{p_i \rightarrow e_i\} : \mathcal{R}) \sigma = \text{let } (e'_i, \sigma_i) = \mathcal{D}[[[p_i/x]e_i]]_{\mathcal{F},\rho} ([p_i/x]\mathcal{R}) \sigma_{i-1} \text{ in } (\text{case } x' \text{ of } \{p_i \rightarrow e'_i\}, \sigma_i) \quad (\text{R28})$$

$$\mathcal{B}[[e']]_{\mathcal{F},\rho} (\text{case } \square \text{ of } \{p_i \rightarrow e_i\} : \mathcal{R}) \sigma = \text{let } (e'_i, \sigma_i) = \mathcal{D}[[e_i]]_{\mathcal{F},\rho} \mathcal{R} \sigma_{i-1} \text{ in } (\text{case } e' \text{ of } \{p_i \rightarrow e'_i\}, \sigma_i) \quad (\text{R29})$$

$$\mathcal{B}[[e']]_{\mathcal{F},\rho} \epsilon \sigma = (e', \sigma) \quad (\text{R30})$$

Figure 2. Two-Driving algorithm

```

| otherwise = Let (NonRec b a) (doBeta env' body as)
  where env' = {inSet = b:inSet env}
doBeta _ fn args = mkApps fn args

```

### 3.1 Application Rule

Rule R4 refers to  $\mathcal{D}_{app}()$ , defined in Figure 4.  $\mathcal{D}_{app}()$  can be inlined in the definition of the driving algorithm, it is merely given a separate name for improved clarity of the presentation.

To ensure termination, we use a variation of the homeomorphic embedding relation  $\trianglelefteq^*$  (?) to define a predicate called “the whistle”. The intuition is that when  $e \trianglelefteq^* f$ ,  $f$  contains all subterms of  $e$ , possibly embedded in other terms. For any infinite sequence  $e_0, e_1, \dots$  there exists  $i$  and  $j$  such that  $i < j$  and  $e_i \trianglelefteq^* e_j$ . Polymorphic recursion might give rise to infinitely many new terms, and hence we consider all types  $\varphi$  equivalent. This condition is sufficient to ensure termination.

Whenever the whistle blows, our algorithm splits the input expression into strictly smaller terms that are transformed separately in the empty context. This might expose new folding opportunities, and allows the algorithm to remove intermediate structures in subexpressions.

## 4. The Homeomorphic Embedding

### 4.1 Invariants

For any infinite chain  $t_1, t_2, \dots$  created from the finite set  $T$ , there exists  $i$  and  $j$  such that  $t_i \trianglelefteq^* t_j$ .

### 4.2 Parallel Homeomorphic Embedding Test

The homeomorphic embedding test can trivially be done in parallel, and since we need all the candidates that we might generalise against there is no risk of doing unnecessary work. This saves 25 seconds on the first digits of e benchmark in nofib.

### 4.3 Only Whistle on Terms With the Same Head

We can considerably cut down on the number of terms to test against if we only test against terms with the same head. This can not be applied recursively.

### 4.4 The Representation Problem

Do we think that  $flip$  ( $flip$   $t$ ) is embedded in

```

case flip y of
  Leaf a → Leaf a
  Branch l r → Branch (flip r) (flip l)

```

#### 4.4.1 Leuschel’s Form

The two terms are  $flip(flip(x))$  and  $Branch(flip(l), flip(r))$ . They are not embedded.

#### 4.4.2 Curried Form

The two terms are  $App(flip, App(flip, x))$  and  $App(App(Branch, App(flip, l)), App(flip, r))$ . The outermost applications matches, the rightmost part of them are homeomorphically embedded, and all we have to do is to find out if  $flip$  is embedded in  $App(Branch, App(flip, l))$ , which it is.

#### 4.4.3 Context Form

The terms are  $[e_1]\mathcal{R}$  and  $[e_2]\mathcal{R}'$ .

1.  $x \trianglelefteq y$
2.  $e_1 = e_2$  and  $\mathcal{R}$  and  $\mathcal{R}'$  are lock-step embedded.
3.  $[e_1]\mathcal{R} \trianglelefteq \text{subterms } \mathcal{R}'$

## 5. Splitting Expressions

### 5.1 Invariants

(Arguments reversed).

$split(t_1, t_2) = (t_g, \sigma)$  such that  $\sigma(t_g) = t_1$  and  $|t_g| < |t_1|$ .

Split tries to preserve as much structure between its input and output by using the most specific generalisation. If this fails (msg returns a variable) split will peel off the outermost part of the term and return the remaining parts as the substitution  $\sigma$ .

Split is always called on two terms on the form  $\mathcal{R}\langle F_1 \rangle$  and  $\mathcal{R}\langle F_2 \rangle$ .

### 5.2 New Msg

Step through the term and use match to find instances, and fail otherwise. This is cheaper than doing the homeomorphic embedding on the entire list of previously seen terms and selecting the best out of that.

### 5.3 New Split

View  $\mathcal{R}\langle F \rangle$  as  $[F] \mathcal{R}_1 : \dots : \mathcal{R}_{n-1} : \mathcal{R}_n : \epsilon$ . Split gives two terms:  $[F] \mathcal{R}_1 : \dots : \mathcal{R}_{n-1} : \epsilon$  and  $[x] \mathcal{R}_n : \epsilon$  where  $x$  is fresh.

### 5.4 Which Term to Generalise Against

We need the notion of best match since the tree benchmark from the language shootout gives two candidates that it can fold against. If we fold against the wrong candidate there is no fusion.

## 6. Invariants

### 6.1 The Driving Algorithm

The driving algorithm is total, terminating, and returns a new term that is an improvement over the input.

## 7. Performance

### 7.1 Sharing New Function Definitions

It is quite common for the case of case-rule to fire. A typical case looks something like this:

```

case ( case x of
  p1 → x1
  ...
  pn → xn) of
[] → []
(x' : xs') → append xs' zs

```

Once the case of case-rule is done the code is:

```

case x of
p1 → case x1
      [] → []
      (x' : xs') → append xs' zs
...
pn → case xn
      [] → []
      (x' : xs') → append xs' zs

```

The driving algorithm will transform the call to append twice, and create two functions that are both isomorphic to append. Creating the functions at the top level and calling the same function from both branches saves both transformation work and reduces code size.

Using a state monad for the driving algorithm, and have that monad store a memoization list augmented with the function definitions will allow the second branch to just insert a call to this new function created in the first branch, thereby saving a lot of transformation effort.

$$\mathcal{D}_{app}(F)_{\mathcal{F},\rho} \mathcal{R} \sigma = \text{let } (\bar{e}', \sigma') = \mathcal{D}[\bar{e}]_{\mathcal{F},\rho} \in \sigma_1 \quad \text{if } \exists(h, \bar{y}, e_1) \in \rho. [\bar{e}/\bar{y}]e_1 = \mathcal{R}\langle F \rangle \quad (1)$$

$$\mathcal{D}_{app}(F)_{\mathcal{F},\rho} \mathcal{R} \sigma = \text{let } (\bar{f}', \sigma_i) = \mathcal{D}[\bar{f}]_{\mathcal{F},\rho} \in \sigma_{i-1} \quad \text{if } \exists(h, \bar{y}, e_i) \in \rho. e_i \leq^* \mathcal{R}\langle F \rangle \quad (2)$$

$$\mathcal{D}_{app}(F)_{\mathcal{F},\rho} \mathcal{R} \sigma = \text{let } (f'_g, \sigma_2) = \mathcal{D}[f_g]_{\mathcal{F},\rho} \in \sigma_i$$

$$\text{in } ([\bar{f}'/\bar{y}]f'_g, \sigma_2)$$

$$\text{let } (e', \sigma_1) = \mathcal{D}[e]_{\mathcal{F},\rho'} \mathcal{R} \sigma$$

$$\text{in } (h \bar{x}, \sigma_1 \cup (h, \lambda \bar{x}. e')) \quad (3)$$

where  $(F = e) \in \mathcal{F}$ ,  
 $\rho' = \rho \cup (h, \bar{x}, \mathcal{R}\langle F \rangle)$ ,  
 $h$  fresh,  
 $\bar{x} = \text{fv}(\mathcal{R}\langle F \rangle)$ ,  
 $(f_g, \bar{f}, \bar{y}) = \text{split}(\mathcal{R}\langle F \rangle, e_1)$

Figure 4. Driving of applications

## 8. Random Notes

### 8.1 Implementation: Name Capture

Any implementation needs to deal with this, but previous articles barely mentions it except in short remarks.

### 8.2 Making a Stupid Supercompiler

Supercompilation is a complex algorithm. We can simplify parts of it and increase the potential sharing of functions for the entire program if we are supercompiling multiple functions. This suggests a post-phase that cleans up after the supercompiler though, which we discuss in Section 8.6.

### 8.3 Evaluation Contexts vs Other Contexts

We have a chance of good things happening with interaction between non-empty contexts  $\mathcal{R}$  and a function in the hole. No such luck with an arbitrary context that has the hole outside the focus for evaluation.

### 8.4 Lambda-lift to Avoid Code Explosion

case (case ... of alts) of (p, q) - $\lambda$  BIG = letrec f p q = BIG in case (case ... of alts) of (p, q) - $\lambda$  f p q

### 8.5 Boring Contexts

Should we specialise functions in boring contexts?  
 map f xs is boring. map fst xs is not. case map f xs of alts is not.

### 8.6 Three Stage Supercompilation?

Having a pre-phase that lambda-lifts big expressions out from case branches and similar constructs, as described in Section 8.4, will give “better” input for the supercompiler to work on if code size is a concern.

A post-phase that inlines non-recursive functions that are only used once will compensate for the less clever supercompiler we have.

### 8.7 What if the Supercompiler is too Stupid?

How many function definitions do we want from append “abc” xs?

### 8.8 Threading Sigma, But Splitting Rho

Letting the folding mechanism call the same function when it is possible saves work during the transformation since the algorithm can immediately fold, and reduces total code size.

However, if we fold too eagerly this will miss out on specialisation opportunities and this will make the final program perform worse than necessary. An example is `append (append xs ys) zs`, which when transformed will create a function isomorphic to `append` itself in the first branch. The recursive call to `append (append xs' ys) zs`

later during the transformation can fold against `append ys zs`, but this will prevent any deforestation from happening.

We therefore only look for renamings, not instances, when trying to fold against things in Sigma. If there exists a specialised version of our current function it will use that, and otherwise it will create a new specialised copy.

Storing the size of the term in Rho allows us to prune the list of possible candidates: a renaming has to be of the same size as current term. Also storing the function head gives more possibility to prune things early when there is no possible match. This turned out to have quite drastic performance implications: the pruning saved 20 seconds on the first benchmark that computes the digits of  $e$  in the `nofib` benchmark suite.

### 8.9 Msg of Typed Terms

Given  $f @ Int (x :: [Int])$  and  $f @ Double (x :: [Double])$ , how do we split it? For now: let the msg fail, and do the ordinary split instead.

### 8.10 Examples

See Figure ??

## Acknowledgments

The authors would like to thank Colin Runciman and Neil Mitchell for valuable discussions about supercompilation. We would also like to thank Simon Marlow for answering countless questions about GHC.