JFP **13** (3): 509–543, May 2003.
DOI: 10.1017/S0956796802004604 Printed in the United Kingdom

*Specification and correctness of lambda lifting**

ADAM FISCHBACH and JOHN HANNAN
Department of Computer Science and Engineering,
The Pennsylvania State University, University Park, PA 16802, USA
(*e-mail:* {fischbac, hannan}@cse.psu.edu)

Abstract

We present a formal and general specification of lambda lifting and prove its correctness with respect to a call-by-name operational semantics. We use this specification to prove the correctness of a lambda lifting algorithm similar to the one proposed by Johnsson. Lambda lifting is a program transformation that eliminates free variables from functions by introducing additional formal parameters to function definitions and additional actual parameters to function calls. This operation supports the transformation from a lexically-structured functional program into a set of recursive equations. Existing results provide specific algorithms and only limited correctness results. Our work provides a more general specification of lambda lifting (and related operations) that supports flexible translation strategies, which may result in new implementation techniques. Our work also supports a simple framework in which the interaction of lambda lifting and other optimizations can be studied and from which new algorithms might be obtained.

1 Introduction

Lambda lifting is a program transformation that eliminates free variables from functions by introducing additional formal parameters to function definitions and additional actual parameters to function calls. The operation finds application in the implementation of functional languages, where functions without free variables can be implemented more easily than those with free variables (Johnsson, 1985; Clinger & Hansen, 1994). Another application for lambda lifting is partial evaluation, where recursive equations (the result of completely lambda lifting a program) provide a convenient representation (Bondorf & Danvy, 1991).

In general, lambda lifting and its inverse, lambda dropping (Danvy & Schultz, 2000), are operations that modify the way in which the implementation of a function accesses the variables occurring in the body of the function and, consequently, the representation of data, including parameters and closures, used by the implementation. The particular choice of lexical structure of a program has no significant importance with respect to the meaning of a program, and we often assume that it has no consequence on the efficiency of the program. Lambda lifting and dropping,

* This material is based upon work supported by the National Science Foundation under Grant No. 9900918.

in part, provide the flexibility that allows an implementation to be indifferent to the choice of structure made by the programmer.

The essential aspects of lambda lifting can be summarized by the following two operations:

1. Remove free variables from functions by inserting additional parameters into the definitions of these functions;
2. Apply these lifted functions to these additional parameters.

Descriptions of lambda lifting, as originally presented by Johnsson (1985) and Hughes (1982), and later by Peyton Jones (1987) and Peyton Jones & Lester (1992), start with these simple concepts and then use algorithms, based on a kind of flow analysis, to fill in the details.

Because these are algorithms, they make specific decisions regarding which variables to lift from a function (all free variables occurring in the function body) and where to lift applications (at each occurrence of the function name). These decisions reflect both the practical considerations of lambda lifting (in the context of compilation) and the limitations of a simple flow-based approach. These works do not provide general principles of lambda lifting from which specific algorithms can be derived and proved correct. They also do not accommodate different design choices (regarding what to lift and where to lift it) and their implications.

Our goal is to provide a foundation, via a high-level, declarative specification, for lambda lifting and related operations. Such a presentation should be devoid of particular implementation or algorithmic decisions. Instead it should support the justification of any operation reasonably based on the informal description given by the two statements above. Proving this specification correct (with respect to a semantics for the language) justifies any operation or algorithm which conforms to this specification. That is, for any given algorithm, we need only prove it correct with respect to the specification, rather than with respect to the semantics of the language.

We wish not only to explicate existing notions of lambda lifting, but also to explore alternatives which might provide better solutions in some applications. We intend to provide a specification which allows for experimenting with flexible strategies of lambda lifting. Also, we wish to provide a framework for exploring the interaction of lambda lifting and related operations including unCurrying (Hannan & Hicks, 2000), closure conversion (Hannan, 1995), arity raising (Hannan & Hicks, 1998), and useless-variable elimination (Fischbach & Hannan, 2001).

The rest of the paper is organized as follows. In the next section we introduce a simple functional language and the basic concepts of lambda lifting. In section 3 we give a formal specification of lambda lifting as a deductive system. In section 4 we prove the correctness of our specification with respect to the type system and operational semantics of the language. In section 5 we present a lambda lifting algorithm similar to the one proposed by Johnsson and use our specification to prove it correct. In section 6 we enhance our specification by introducing dependent types for parameter lifting. Introducing polymorphic types into our language is discussed in section 7, and in section 8 we conclude.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Sigma(c) = \tau}{\Gamma \vdash c : \tau}$$

$$\frac{\Gamma\{x : \tau_1\} \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \mathbin{@} e_2 : \tau}$$

$$\frac{\Gamma^* = \Gamma\{\overline{f_i{:}\tau_i}\} \quad \Gamma^* \vdash e_i : \tau_i \quad \Gamma^* \vdash e : \tau \quad i \in \{1..n\}}{\Gamma \vdash \mathsf{letrec}\ \overline{f_i = e_i}\ \mathsf{in}\ e : \tau}$$

Fig. 1. Type system.

2 Overview of lambda lifting

Lambda lifting is a transformation on lexically-scoped programs that provides a means for eliminating free variables from function definitions. The operation was developed independently by Hughes (1982) and Johnsson (1985), both in the context of compiling functional languages. Peyton Jones later provided a careful development of the operation (1987) and Peyton Jones & Lester (1992) in the larger context of language implementation. Lambda dropping, the inverse of lambda lifting was more recently proposed by Danvy & Schultz (2000) as an operation to restore the lexical block structure of programs following lambda lifting. In all cases the presentation of these operations is mostly algorithmic and restricted in their application to higher-order functions. With respect to correctness, only some preliminary results by Danvy (1998) exist.

We present here an overview of the operations of lambda lifting and dropping described in these works, but we do not discuss the algorithms given there.

2.1 The language

We present our specification of lambda lifting for a small higher-order functional language. The grammar for expressions and types for this language is

$$e \quad ::= \quad c \mid x \mid \lambda x.e \mid e_1 \mathbin{@} e_2 \mid \mathsf{letrec}\ \overline{f_i = e_i}\ \mathsf{in}\ e$$

$$\tau \quad ::= \quad \iota \mid \tau \to \tau$$

Both x and f are expression variables in our language. We let c range over the set of pre-defined constants. We use the @ symbol to explicitly represent function application. Mutually recursive functions can be defined using letrec. Throughout the paper we use the notation $\overline{f_i = e_i}$ to represent the simultaneous declarations $f_1 = e_1 \ldots f_n = e_n$. We let ι and $\tau \to \tau$ range over base types and function types, respectively.

The type system given in figure 1 axiomatizes the judgment $\Gamma \vdash e : \tau$ relating an expression e to its simple type τ given a context Γ mapping variables to types. We assume a pre-defined signature Σ mapping each constant to its type. In the rule for letrec, $\Gamma\{\overline{f_i : \tau_i}\}$ extends the context Γ to include mappings for all f_i declared in the letrec, and n is the number of simultaneous declarations.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

$$\frac{}{c \hookrightarrow c}$$

$$\frac{}{\lambda x.e \hookrightarrow \lambda x.e}$$

$$\frac{e_1 \hookrightarrow \lambda x.e \quad e[e_2/x] \hookrightarrow v}{e_1 \mathbin{@} e_2 \hookrightarrow v}$$

$$\frac{e[\overline{(\mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ e_j)/f_j}] \hookrightarrow v \quad i, j \in \{1..n\}}{\mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ e \hookrightarrow v}$$

Fig. 2. Operational semantics.

We use a call-by-name operational semantics for our language. The rules in figure 2 axiomatize the judgment $e_1 \hookrightarrow e_2$, which reads "expression e_1 evaluates to e_2." The notation $e[e'/x]$ represents the operation of substituting expression e' for each occurrence of free variable x in e. For mutual recursion, it is necessary to replace (in e) each occurrence of the variable f_j with $\mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ e_j$, for all f_j declared in the letrec. We use a substitution-based semantics as opposed to one involving variable environments and function closures in order to facilitate the proof of operational correctness in section 4.

In this paper, we consider only a call-by-name language, in keeping with the original presentation of lambda lifting. However, call-by-value languages can also benefit from lambda lifting by reducing the number of variables in closures.

2.2 The basics of lambda lifting

Lambda lifting has been described as a two step process, based on Johnsson's algorithm (Danvy & Schultz, 2000):

1. *Parameter Lifting*. Free variables of a function are eliminated by introducing additional parameters to the function. Call sites of the function are correspondingly supplied with these variables as additional arguments.
2. *Block Floating*. Local function definitions with no free variables can be floated outwards through the block structure of the program until they become global definitions.

The block floating step is trivial once all free variables have been eliminated from functions. More generally, function definitions (possibly containing free variables) can be floated outwards (or inwards) as long as the function is not moved (outwards) outside the scope of the free variables or moved (inwards) inside the scope of a declaration of a variable with the same name as a free variable occurring in the function. The practical aspects of block floating have been studied by Peyton Jones *et al.* (1996). The correctness of this operation is nearly trivial, and we do not address it here. Our results do not support any new notions of block floating and so we do not discuss it further.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

The parameter lifting step is a more subtle operation and care must be taken to ensure correctness. When we insert a lifted (actual) parameter x, we must ensure that x is in the scope of its intended declaration (and not shadowed by another declaration of x).

Consider the following expression. In this and subsequent examples we use the expression `let x = e1 in e2 end` for the beta redex `(λx.e2) @ e1` for readability. We also assume that our language supports the addition operation for the presentation of the examples. Lifted variables are identified by bold-italicized type.

```
let x = e0
in letrec f = λy.y + x
   in let x = e1
      in f @ x end
   end
end
```

Naively applying the description of parameter lifting above yields the following incorrect translation:

```
let x = e0
in letrec f = λx.λy.y + x
   in let x = e1
      in f @ x @ x end
   end
end
```

The lifted parameter has been captured by a subsequent declaration of `x`.

Consider a second example:

```
let x = e0
in letrec f = λy.y + x;
          h = λk.k @ 5
   in h @ f end
end
```

Existing lambda lifting strategies require the lifted parameters to be inserted at each occurrence of the lifted function, giving the following as the only possible translation:

```
let x = e0
in letrec f = λx.λy.y + x;
          h = λk.k @ 5
   in h @ (f @ x) end
end
```

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

However, in this simple example we can see that the following is a correct translation:

```
let x = e0
in letrec f = λx.λy.y + x;
          h = λx.λk.k @ x @ 5
   in h @ x @ f  end
end
```

At this point in the discussion we are not concerned whether this is a useful alternative, only that it is a *possible* translation. We want to ensure that our specification can handle such a translation.

Such handling of higher-order functions is fraught with pitfalls, however. Consider another example:

```
let g = let x = e0
        in letrec f = λy.y + x
           in f end
in g @ e1
end
```

Naively performing parameter lifting which delays the application of `f` to its lifted parameter yields incorrect results:

```
let g = let x = e0
        in letrec f = λx.λy.y + x
           in f end
in g @ x @ e1
end
```

The lifted parameter ***x*** has escaped its scope. Simply being in scope is not sufficient since it could be the wrong scope. Consider the expression:

```
let g = let x = e0
        in letrec f = λy.y + x
           in f end
in let x = e2
   in g @ e1
end
```

which, again naively, could be incorrectly lifted to

```
let g = let x = e0
        in letrec f = λx.λy.y + x
           in f end
in let x = e2
   in g @ x @ e1
end
```

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

even though the lifted parameter is inserted in a context in which it is in the scope of a declaration for x.

The problems of such higher-order lifting are further complicated by the need to ensure that all constraints of lifted functions are met. Consider the following example:

```
let x = e0
in letrec f = λy.y + x;
          g = λy.y + 3;
          h = λk.k @ 5
   in (h @ f) + (h @ g) end
end
```

In this case, lifting f and delaying the application to ***x*** until the body of h is possible, but only if we also perform the vacuous lifting of ***x*** from g:

```
let x = e0
in letrec f = λx.λy.y + x;
          g = λx.λy.y + 3;
          h = λx.λk.k @ x @ 5
   in (h @ x @ f) + (h @ x @ g) end
end
```

The problem in the first example can easily be avoided by assuring that no variable declaration occurs in the scope of a variable of the same name. The problems in the other examples have not been an issue in previous descriptions of lambda lifting because the algorithms given in those works require that if f is the name of the lifted function then f must be applied directly to the lifted parameters. As suggested by these examples, extending previous work to handle higher-order functions in the manner above requires specifying and solving constraints among function declarations and calls.

In describing lambda lifting of higher-order programs Danvy & Schultz (2000) limit themselves to first-order function applications. They state that extending the operation to higher-order function applications would require a control-flow analysis (Shivers, 1991). However, a type-based analysis provides a suitable, and perhaps preferable, framework for incorporating higher-order features. This is the kind of analysis we present in section 3.

2.3 Lambda dropping

The inverse of lambda lifting has been called lambda dropping by Danvy & Schultz (2000). As the inverse of lifting, they describe it via two steps: block sinking and parameter dropping. They provide a definition of dropped programs and give an algorithm for lambda dropping. While the relation to lambda lifting is apparent, the authors only conjecture that lifting and dropping are inverses. Because previous works have presented lambda lifting and dropping algorithmically, this inverse correspondence has been obscured. If, instead, a specification of lambda lifting is

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

given as a binary relation (between an input term and a lifted form of the term), then the specification also describes lambda dropping.

3 Specification of parameter lifting

We give a formal specification of parameter lifting as a deductive system that axiomatizes a relation between two terms. The second term is a parameter-lifted form of the first. (Equivalently, the first term is a parameter-dropped form of the second.) As our goal is to provide a general description of these operations we will not enforce any particular strategy of which parameters to lift or where to lift them. The system is non-deterministic in the sense that a given term can possibly be related to many lifted forms. We use type information to provide constraints between terms and to direct the definition of the relation between terms. Most of the inference rules follow the structure of a traditional type system for simple types, but the specification also contains additional rules unique to the problem of lambda lifting.

3.1 Singleton types

Because every lifting of a parameter in a function declaration must be accompanied by the appropriate application of a term to that parameter, we need to generate constraints between terms. As already demonstrated, the names of variables have particular importance in these constraints and we must be careful with names. We extend the traditional definition of simple types we presented in section 2 to include singleton types (Aspinall, 1995) that provide information regarding lifted parameters:

$$\tau \quad ::= \quad \iota \mid \tau \to \tau \mid \{e\}_\tau \to \tau$$

The type $\tau_1 \to \tau_2$ denotes the traditional function type, while the type $\{e\}_{\tau_1} \to \tau_2$ denotes the type of a function obtained by lifting the expression e out of the body of a function of type τ_2. The only term that inhabits the singleton type $\{e\}_{\tau_1}$ is the expression e of type τ_1. In the grammar we have presented here, e is an arbitrary expression. The lambda lifting specification we present in the following subsection restricts this expression to be a variable (i.e. only variables can be lifted). However, the more general specification in section 4 does allow the lifting of arbitrary expressions, which enables full laziness (Peyton Jones, 1987).

3.2 Parameter lifting

The specification in figure 3 axiomatizes the relation for parameter lifting. The specification uses a judgment $\Gamma \rhd e : \tau \Rightarrow e'$ in which e and e' are terms, Γ is a context mapping variables to types, and τ is a type that may include singleton types. We read this judgment as stating that under the assumption of Γ, expression e can be lifted to e' of type τ. We assume that the signature Σ does not contain any singleton types.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

$$\frac{\Gamma(x) = \tau}{\Gamma \rhd x : \tau \Rightarrow x} \text{ (var)} \qquad \frac{\Sigma(c) = \tau}{\Gamma \rhd c : \tau \Rightarrow c} \text{ (const)}$$

$$\frac{\Gamma\{y{:}\tau_1\} \rhd e : \tau \Rightarrow e' \quad y \notin \mathsf{dom}(\Gamma) \quad FV(\tau_1 \to \tau) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \rhd \lambda y.e : \tau_1 \to \tau \Rightarrow \lambda y.e'} \text{ (abs)}$$

$$\frac{\Gamma(x) = \tau_1 \quad \Gamma \rhd \lambda y.e : \tau \Rightarrow \lambda z.e'}{\Gamma \rhd \lambda y.e : \{x\}_{\tau_1} \to \tau \Rightarrow \lambda x.\lambda z.e'} \text{ (lift-abs)}$$

$$\frac{\Gamma \rhd e_1 : \tau_1 \to \tau \Rightarrow e_1' \quad \Gamma \rhd e_2 : \tau_1 \Rightarrow e_2'}{\Gamma \rhd e_1 \mathbin{@} e_2 : \tau \Rightarrow e_1' \mathbin{@} e_2'} \text{ (app)}$$

$$\frac{\Gamma \rhd e : \{x\}_{\tau_1} \to \tau \Rightarrow e' \quad \Gamma(x) = \tau_1}{\Gamma \rhd e : \tau \Rightarrow e' \mathbin{@} x} \text{ (lift-app)}$$

$$\frac{\begin{array}{c}\Gamma^* = \Gamma\{\overline{f_i : \tau_i}\} \quad \Gamma^* \rhd e_i : \tau_i \Rightarrow e_i' \\ \Gamma^* \rhd e : \tau \Rightarrow e' \quad f_i \notin \mathsf{dom}(\Gamma) \quad FV(\tau) \subseteq \mathsf{dom}(\Gamma) \quad i \in \{1..n\}\end{array}}{\Gamma \rhd \mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ e : \tau \Rightarrow \mathsf{letrec}\ \overline{f_i = e_i'}\ \ \mathsf{in}\ e'} \text{ (letrec)}$$

Fig. 3. Parameter lifting.

The rules (const), (var), (app), (abs), and (letrec) are simple extensions to the traditional typing rules in figure 1. The types in these rules do range over the types defined above, but make no use of singleton types. (In the rules (app) and (abs), however, τ_1 may not be a singleton type.) The rules (abs) and (letrec) do include constraints to ensure that the introduction of a variable name does not shadow an existing declaration of the same name. (This ensures that we avoid one of the problems introduced in section 2.) These two rules also include conditions to ensure that lifted parameters (occurring free in types) do not escape their scope. We use $FV(\tau)$ to denote the set of expression variables appearing as singleton types in τ.

Only the rules (lift-abs) and (lift-app) make explicit use of singleton types, and these are the rules that introduce formal parameters and actual parameters, respectively. The rule (lift-abs) supports the parameter lifting of any variable x that is currently in scope ($x \in \mathsf{dom}(\Gamma)$). The argument type of the resulting expression is a singleton type including the name of the lifted parameter. In the conclusion of (lift-abs) the subject expression is a function, and this ensures that we only parameter lift at the point of function definitions. Observe that this same subject expression appears in the antecedent of the rule. This allows multiple parameters to be lifted from a function definition (which is the reason why we use z instead of y in the translated term).

The rule (lift-app) complements (lift-abs) by supporting the introduction of a new application to any term of the appropriate type. The rule ensures that $x \in \mathsf{dom}(\Gamma)$ to avoid one of the problems illustrated in section 2. Observe that the rule (lift-app)

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

allows any term of type $\{x\}_{\tau_1} \to \tau$ (not just a variable) to be applied to lifted parameters.

3.3 Examples

Some simple examples illustrate the kinds of parameter lifting supported by this specification. Consider the term

```
let x = e1
in letrec f = λy.y + x
   in letrec g = λh.λz.h @ (h @ z)
      in g @ f @ e2 end
   end
end
```

Using our specification, we can parameter lift this expression to the following:

1. We can immediately apply the lifted function to the parameter `x`:

```
let x = e1
in letrec f = (λx.λy.y + x) @ x
   in letrec g = λh.λz.h @ (h @ z)
      in g @ f @ e2 end
   end
end
```

2. We can apply `f` to the parameter x:

```
let x = e1
in letrec f = λx.λy.y + x
   in letrec g = λh.λz.h @ (h @ z)
      in g @ (f @ x) @ e2 end
   end
end
```

3. We can apply `h` (which is bound to `f`) to the parameter x:

```
let x = e1
in letrec f = λx.λy.y + x
   in letrec g = λh.λz.(h @ x @ (h @ x @ z))
      in g @ f @ e2 end
   end
end
```

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

4. We can apply h (which is bound to f) to the parameter ***x***, and also lift ***x*** from g:

```
let x = e1
in letrec f = λx.λy.y + x
   in letrec g = λx.λh.λz.h @ x @ (h @ x @ z)
      in g @ x @ f @ e2 end
   end
end
```

The first translation is the initial method proposed by Johnsson (1985), which he immediately rejects as useless. The second translation is what is typically performed by existing parameter lifting algorithms, including those proposed by Johnsson (Johnsson, 1985), Peyton-Jones (1987) and Danvy (2000). The third and fourth translations are, to our knowledge, not supported in general by any existing algorithms. Just as our previous work using type-based systems have extended results to support higher-order analyses and translations (Hannan & Hicks, 1998, 2000), our specification of parameter lifting also benefits from the nature of type systems. The argument for choosing the third or fourth translation over the second translation can arise due to consideration of unCurrying. If our language supported unCurrying, we could unCurry the definition of f (and also g) and the corresponding applications:

```
let x = e1
in let f = λ[x,y].y + x
   in let g = λ[x,h,z].h @ [x,h @ [x,z]]
      in g @ [x,f,e2] end
   end
end
```

This unCurrying is not possible using the second translation.

In the presence of separate compilation, we must ensure that no parameters are lifted out of any function that is visible outside of the current compilation unit. This restriction can be enforced in our system by preventing singleton types from appearing in the types of exportable functions.

4 Correctness

Having specified lambda lifting we now demonstrate its correctness by providing appropriate relationships between our specification and static and dynamic (operational) semantics for the language.

4.1 Type correctness

We show that the system in figure 3 derives judgments over exactly the terms typable by the rules in figure 1. Additionally, we show that every typable source term can be related to some target term.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

To prove these properties we first define two translations from the types defined in section 3 (including singleton types) to the types defined in section 2 (not including singleton types):

Definition 1
For any type τ, let $|\tau|$ and $||\tau||$ be defined as

$$\begin{array}{lcllcl} |\iota| & = & \iota & ||\iota|| & = & \iota \\ |\tau_1 \to \tau_2| & = & |\tau_1| \to |\tau_2| & ||\tau_1 \to \tau_2|| & = & ||\tau_1|| \to ||\tau_2|| \\ |\{x\}_{\tau_1} \to \tau_2| & = & |\tau_2| & ||\{x\}_{\tau_1} \to \tau_2|| & = & ||\tau_1|| \to ||\tau_2|| \end{array}$$

The type $||\tau||$ is the erasure of τ and the type $|\tau|$ removes the lifting information from τ. For type contexts, if $\Gamma(x) = \tau$ then $||\Gamma||(x) = ||\tau||$ and $|\Gamma|(x) = |\tau|$.

Theorem 1 (*Type Completeness*)
If $\Gamma \rhd e : \tau$ then there exists a term e' such that $\Gamma \rhd e : \tau \Rightarrow e'$.

The proof follows by constructing a deduction which performs no parameter lifting (no uses of (lift-abs) or (lift-app)).

We additionally have that the specification relates only typable terms.

Theorem 2 (*Type Correctness*)
If $\Gamma \rhd e : \tau \Rightarrow e'$ then $||\Gamma|| \rhd e' : ||\tau||$ and $|\Gamma| \rhd e : |\tau|$.

The proof can be found in Appendix A.

Type correctness tells us that we are, at least, constructing expressions which satisfy the constraints given by the type system. We still need to demonstrate that operationally, a parameter-lifted expression is equivalent to the original expression.

4.2 Operational correctness

Before presenting the theorem of operational correctness, we present our lambda lifting specification using a substitution-based semantics (i.e. without explicit contexts).

The rules in figure 4 axiomatize the judgment $e : \tau \Rightarrow e'$ where e' has type τ and is a lambda-lifted form of e. The (app) rule is nearly identical to that in figure 3, but the others require some comment.

Rather than using an explicit variable rule, we use hypothetical assumptions (introduced by (abs) and (letrec)) in order to reason about variables. In the rule (abs), the variables c and c' are substituted for the variables y and y' in the source and target terms, respectively. The universal quantification guarantees that c and c' are fresh variables, thus avoiding the need for the two constraints required for (abs) in figure 3. Since these variables are quantified only over the antecedent of the rule, c' cannot appear in the type $\tau \to \tau_1$ and hence cannot be lifted outside of its scope.

The (letrec) rule is similar to (abs) except that it must allow for mutual recursion. Because of the universal quantification, no additional constraints are required here either.

In figure 3, the (lift-app) rule requires that the lifted variable occur in the context.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

$$\frac{e_1 : \tau_2 \to \tau \Rightarrow e_1' \quad e_2 : \tau_2 \Rightarrow e_2'}{e_1 \mathbin{@} e_2 : \tau \Rightarrow e_1' \mathbin{@} e_2'} \text{ (app)}$$

$$\frac{e_2 : \tau_1 \Rightarrow e_2' \quad e_1 : \{e_2'\}_{\tau_1} \to \tau \Rightarrow e_1'}{e_1 : \tau \Rightarrow e_1' \mathbin{@} e_2'} \text{ (lift-app)}$$

$$\frac{\forall c \forall c' (c : \tau \Rightarrow c' \supset e[c/y] : \tau_1 \Rightarrow e'[c'/y'])}{\lambda y.e : \tau \to \tau_1 \Rightarrow \lambda y'.e'} \text{ (abs)}$$

$$\frac{\lambda y.e : \tau \Rightarrow \lambda z.m[e_1'/x'] \quad e_1 : \tau_1 \Rightarrow e_1' \quad x : \tau_1 \Rightarrow x'}{\lambda y.e : \{e_1'\}_{\tau_1} \to \tau \Rightarrow \lambda x'.\lambda z.m} \text{ (lift-abs)}$$

$$\frac{\begin{array}{c}\forall c_i, c_i' (c_i : \tau_i \Rightarrow c_i' \supset e_j[\overline{c_i/f_i}] : \tau_j \Rightarrow e_j'[\overline{c_i'/f_i'}]) \\ \forall c_i, c_i' (c_i : \tau_i \Rightarrow c_i' \supset e[\overline{c_i/f_i}] : \tau \Rightarrow e'[\overline{c_i'/f_i'}]) \quad i, j \in \{1..n\}\end{array}}{\mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ \ e : \tau \Rightarrow \mathsf{letrec}\ \ \overline{f_i' = e_i'}\ \ \mathsf{in}\ \ e'} \text{ (letrec)}$$

Fig. 4. Parameter lifting without contexts.

The same requirement is enforced in the corresponding rule in figure 4 by making sure that a translation of the lifted term exists.

Finally, (lift-abs) allows any expression to be lifted out of the body of an abstraction provided that a translation for that expression exists. In fact, this rule allows *any* expression to be lifted, not just variables. This is, in fact, more general than the specification in figure 3. A translated term e' is lifted out of the body of the abstraction and replaced with some variable x'. This variable is bound in the translated abstraction and e' is included in the singleton type. So, the function must later be applied to the expression e'.

Theorem 3

For all closed terms e, if $\cdot \rhd e : \tau \Rightarrow e'$ then $e : \tau \Rightarrow e'$ is derivable.

The proof requires a generalization to open terms and a correspondence between open hypotheses and contexts.

Note that because of the generalization of (lift-abs), the converse is not true. However, we do not need an equivalence between the two systems. If the system in figure 4 is operationally correct then, because of the soundness theorem above, the system in figure 3 is operationally correct.

To demonstrate operational correctness we must ensure that the translation of a term preserves the meaning (value) of a term. Since values can be functions that, in the translated case, can contain parameter-lifted terms, we can only expect the values to be related via the translation. Preserving the meaning includes, in general, termination properties. So we must show that one term has a value iff its translation (in either direction) has a value.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

Theorem 4 (*Operational Correctness*)

1. If $e \hookrightarrow v$ and $e : \tau \Rightarrow e'$ are derivable then there exists a v' such that $e' \hookrightarrow v'$ and $v : \tau \Rightarrow v'$ are derivable ;
2. If $e' \hookrightarrow v'$ and $e : \tau \Rightarrow e'$ are derivable then there exists a v such that $e \hookrightarrow v$ and $v : \tau \Rightarrow v'$ are derivable.

The proof can be found in Appendix A.

A simple corollary to the theorem gives us a desired result for closed programs of base type:

Corollary 1
If $e : \iota \Rightarrow e'$ then $e \hookrightarrow v$ iff $e' \hookrightarrow v$.

5 Simple parameter lifting algorithm

Our specification provides a general, high-level description of parameter lifting that supports flexibility in the choice of which parameters to lift. Because of this property, many possible algorithms could be based on this specification. We present in this section one example of a parameter lifting algorithm closely related to Johnsson's algorithm (Johnsson, 1985). We then demonstrate how the specification in figure 3 can be used to prove that this algorithm is correct.

A parameter lifting algorithm must make specific choices as to which parameters to lift and where to insert applications. We choose the following based on Johnsson's approach: all free variables (excluding function names) are lifted from the body of a named function, and each occurrence of a function name is applied directly to that function's lifted parameters.

5.1 Two-phase specification

Following methods successfully employed in previous work (e.g. Hannan & Hicks, 1998, 2000), we first present a two-phase specification for parameter lifting from which a specific algorithm is more easily derivable than from the general specification in figure 3.

The first phase of this specification deduces which variables are lifted and where new applications are inserted. Following Johnsson's algorithm, we avoid lifting function names. The reason for this is that, after block floating, all functions will be global. We will use f to represent function names, and x and y to represent all other variables. We use v when the distinction is irrelevant.

The rules in figure 5 define the judgment $\Lambda \rhd e : (\tau, \theta) \Rightarrow m$ in which e is an input term, τ is the type of e, θ is an ordered set of variables, m is an annotated form of e, and Λ maps variables to types and ordered sets. The annotated term m is the same as the input term e except that all variables and letrec expressions are annotated with sets of lifted parameters. These annotations are then used in the second phase of the specification, which does the actual lifting of parameters.

The set θ represents all free variables in m (except function names), including all annotations. In Λ, if a function name f maps to $(\tau_1 \to \tau_2, \theta)$, then θ contains all the

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

$$\frac{\Sigma(c) = \tau}{\Lambda \rhd c : (\tau, \emptyset) \Rightarrow c} \text{ (const)}$$

$$\frac{\Lambda(x) = (\tau, \emptyset)}{\Lambda \rhd x : (\tau, \{x\}) \Rightarrow x^{\emptyset}} \text{ (param)} \qquad \frac{\Lambda(f) = (\tau, \theta)}{\Lambda \rhd f : (\tau, \theta) \Rightarrow f^{\theta}} \text{ (fun-name)}$$

$$\frac{\Lambda\{y : (\tau_1, \emptyset)\} \rhd e : (\tau, \theta) \Rightarrow m \quad y \notin \mathsf{dom}(\Lambda)}{\Lambda \rhd \lambda y.e : (\tau_1 \to \tau, \theta - \{y\}) \Rightarrow \lambda y.m} \text{ (abs)}$$

$$\frac{\Lambda \rhd e_1 : (\tau_1 \to \tau, \theta_1) \Rightarrow m_1 \quad \Lambda \rhd e_2 : (\tau_1, \theta_2) \Rightarrow m_2}{\Lambda \rhd e_1 \mathbin{@} e_2 : (\tau, \theta_1 \cup \theta_2) \Rightarrow m_1 \mathbin{@} m_2} \text{ (app)}$$

$$\frac{\begin{array}{c}\Lambda^* = \Lambda\{\overline{f_i : (\tau_i, \theta_i)}\} \\ \Lambda^* \rhd \lambda y_i.e_i : (\tau_i, \theta_i) \Rightarrow \lambda y_i.m_i \quad \Lambda^* \rhd e : (\tau, \theta) \Rightarrow m \quad f_i \notin \mathsf{dom}(\Lambda) \quad i \in \{1..n\}\end{array}}{\Lambda \rhd \mathsf{letrec}\ \overline{f_i = \lambda y_i.e_i}\ \mathsf{in}\ e : (\tau, \bigcup \theta_i \ \cup\ \theta) \Rightarrow \mathsf{letrec}\ \overline{f_i =^{\theta_i} \lambda y_i.m_i}\ \mathsf{in}\ m} \text{ (letrec)}$$

Fig. 5. Parameter lifting annotation phase.

$$\frac{}{c \Rightarrow_t c} \text{ (const)} \qquad \frac{}{y^{\emptyset} \Rightarrow_t y} \text{ (var)} \qquad \frac{y^s \Rightarrow_t e}{y^{x::s} \Rightarrow_t e \mathbin{@} x} \text{ (var-app)}$$

$$\frac{m \Rightarrow_t e}{\lambda y.m \Rightarrow_t \lambda y.e} \text{ (abs)} \qquad \frac{m_1 \Rightarrow_t e_1 \quad m_2 \Rightarrow_t e_2}{m_1 \mathbin{@} m_2 \Rightarrow_t e_1 \mathbin{@} e_2} \text{ (app)}$$

$$\frac{m \Rightarrow_t e \quad g_i =^{\theta_i} \lambda y_i.m_i \Rightarrow_t g_i = e_i \quad \text{for } i \in \{1..n\}}{\begin{array}{c}\mathsf{letrec}\ g_1 =^{\theta_1} \lambda y_1.m_1 \cdots g_n =^{\theta_n} \lambda y_n.m_n\ \mathsf{in}\ m \Rightarrow_t \\ \mathsf{letrec}\ g_1 = e_1 \cdots g_n = e_n\ \mathsf{in}\ e\end{array}} \text{ (letrec)}$$

$$\frac{\lambda y.m \Rightarrow_t e}{g =^{\emptyset} \lambda y.m \Rightarrow_t g = e} \text{ (decl)} \qquad \frac{g =^{s} \lambda x \lambda y.m \Rightarrow_t g = e}{g =^{x::s} \lambda y.m \Rightarrow_t g = e} \text{ (decl-lift)}$$

Fig. 6. Parameter lifting translation phase.

parameters lifted from the function f. Note that if a variable x maps to (τ, θ) then θ is empty. Since θ is an *ordered* set, all set operations must be order-preserving.

The rules for variables and letrec in figure 5 are the only rules that actually annotate terms. In the letrec rule, each θ_i contains all the free variables occurring in the corresponding function $\lambda y_i.m_i$. These are all the parameters that will be lifted from the function body. Each function name f_i must then map to, and each declaration within the letrec must be annotated with, the corresponding set θ_i.

The variable rules force each occurrence of a variable to be annotated with its corresponding list of lifted parameters.

The translation phase defines the judgment $m \Rightarrow_t e$ in which m is an annotated term and e is the translated (parameter lifted) form of m. The rules for translation are straightforward and can be found in figure 6.

Again, the only interesting cases are those for variables and letrec. Each variable

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

must be applied to all of the lifted parameters occurring in its annotation. Similarly, all named functions must include bindings for all lifted parameters.

We can prove that this two-phase specification is sound with respect to the more general specification in figure 3. Observe that the two type systems presented utilize different mechanisms for conveying parameter lifting. The original lambda lifting specification characterizes expressions by using annotated types, while the inference-phase specification characterizes expressions by using sets of variables. To express a relationship between judgments in the two systems we first require a relation between these two.

Definition 2
Let $(\tau, \theta) \succ_\Lambda \tau'$ be the least relationship closed under the following:

1. $(\tau, \emptyset) \succ_\Lambda \tau$
2. $(\tau, y::\theta) \succ_\Lambda \tau'$ if $\Lambda(y) = (\tau_y, \emptyset)$ and $(\{y\}_{\tau_y} \to \tau, \theta) \succ_\Lambda \tau'$

Next we define a correspondence between contexts.

Definition 3
Contexts Λ and Γ correspond, written $\Lambda \cong \Gamma$, iff $\mathsf{dom}(\Lambda) = \mathsf{dom}(\Gamma)$ and for all $v \in \mathsf{dom}(\Lambda)$, if $\Lambda(v) = (\tau, \theta)$ and $\Gamma(v) = \tau'$ then $(\tau, \theta) \succ_\Lambda \tau'$.

This relationship focuses on the correspondence between the expression variables occurring in the sets θ and the expression variables occurring in singleton types.

We introduce a notion of *closed* contexts that provides a reasonable (and required) constraint on the sets occurring in contexts.

Definition 4 (*Closed Contexts*)
A context Λ is *closed*, written $\mathsf{Closed}(\Lambda)$, iff for all $x, f \in \mathsf{dom}(\Lambda)$,

1. if $\Lambda(f) = (\tau, \theta)$ then $\theta \subseteq \mathsf{dom}(\Lambda)$, and
2. if $\Lambda(x) = (\tau, \theta)$ then $\theta = \emptyset$.

We need only deal with closed contexts because the free variables in a type (of a function) refer to lifted variables, and these variables must be declared in an enclosing scope, and hence in the context.

We can then state the soundness property of the two-phase system.

Theorem 5
If $\mathsf{Closed}(\Lambda)$, $\Lambda \rhd e : (\tau, \theta) \Rightarrow m$, $m \Rightarrow_t e'$, and $\Lambda \cong \Gamma$ then $\Gamma \rhd e : \tau \Rightarrow e'$.

The proof can be found in Appendix A.

5.2 Example algorithm

We can now define a recursive algorithm based on this two-phase specification and prove its correctness. The algorithm is also partitioned into two phases (figure 7): $\mathscr{PL}$, which annotates the input term, and translate, which introduces variable bindings and applies function names to the function's lifted parameters.

$\mathscr{PL}$ takes a context Δ, which maps variables to ordered sets, and an input term e,

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

$$\mathscr{PL}(\Delta, c) = (\emptyset, c)$$

$$\begin{array}{l}\mathscr{PL}(\Delta, x) = \mathsf{let}\\ \quad \theta = \Delta(x)\\ \mathsf{in}\ (\{x\}, x^{\theta})\end{array}$$

$$\begin{array}{l}\mathscr{PL}(\Delta, f) = \mathsf{let}\\ \quad \theta = \Delta(f)\\ \mathsf{in}\ (\theta, f^{\theta})\end{array}$$

$$\begin{array}{l}\mathscr{PL}(\Delta, \lambda y.e) = \mathsf{let}\\ \quad (\theta, m) = \mathscr{PL}(\Delta\{y : \emptyset\}, e)\\ \mathsf{in}\ (\theta - \{y\}, \lambda y.m)\end{array}$$

$$\begin{array}{l}\mathscr{PL}(\Delta, e_1 \mathbin{@} e_2) = \mathsf{let}\\ \quad (\theta_1, m_1) = \mathscr{PL}(\Delta, e_1)\\ \quad (\theta_2, m_2) = \mathscr{PL}(\Delta, e_2)\\ \mathsf{in}\ (\theta_1 \cup \theta_2, m_1 \mathbin{@} m_2)\end{array}$$

$$\begin{array}{l}\mathscr{PL}(\Delta, \mathsf{letrec}\ \overline{g_i = \lambda y_i.e_i}\ \mathsf{in}\ e) = \mathsf{let}\\ \quad \Delta' = \Delta\{g_1 : \theta_{g_1}, \ldots, g_n : \theta_{g_n}\}\\ \quad (\theta_1, \lambda y_1.m_1) = \mathscr{PL}(\Delta', \lambda y_1.e_1)\\ \qquad\qquad \vdots\\ \quad (\theta_n, \lambda y_n.m_n) = \mathscr{PL}(\Delta', \lambda y_n.e_n)\\ \quad \varepsilon = \mathsf{solve}(\{\theta_{g_1} = \theta_1 \backslash_{\theta_{g_1}}, \ldots, \theta_{g_n} = \theta_n \backslash_{\theta_{g_n}}\})\\ \quad (\theta, m) = \mathscr{PL}(\varepsilon\Delta', e)\\ \mathsf{in}\ (\varepsilon(\theta_{g_1}) \cup \ldots \cup \varepsilon(\theta_{g_n}) \cup \theta,\\ \quad \mathsf{letrec}\ g_1 =^{\varepsilon(\theta_{g_1})} \lambda y_1.\varepsilon m_1 \cdots g_n =^{\varepsilon(\theta_{g_n})} \lambda y_n.\varepsilon m_n\ \mathsf{in}\ m)\end{array}$$

$$\mathsf{translate}(y^{x::s}) = \mathsf{translate}(y^{s}) \mathbin{@} x$$
$$\mathsf{translate}(y^{\emptyset}) = y$$

$$\begin{array}{l}\mathsf{translate}(\mathsf{letrec}\ g_1 =^{\theta_1} \lambda y_1.m_1 \ldots g_n =^{\theta_n} \lambda y_n.m_n\ \mathsf{in}\ m) =\\ \quad \mathsf{letrec}\ \mathsf{translate}(g_1 =^{\theta_1} \lambda y_1.m_1) \ldots \mathsf{translate}(g_n =^{\theta_n} \lambda y_n.m_n)\ \mathsf{in}\ \mathsf{translate}(m)\end{array}$$

$$\mathsf{translate}(g =^{x::s} \lambda y.m) = \mathsf{translate}(g =^{s} \lambda x.\lambda y.m)$$
$$\mathsf{translate}(g =^{\emptyset} \lambda y.m) = (g = \mathsf{translate}(\lambda y.m))$$

$$\mathsf{translate}(\lambda y.m) = \lambda y.\mathsf{translate}(m)$$
$$\mathsf{translate}(m_1 \mathbin{@} m_2) = \mathsf{translate}(m_1) \mathbin{@} \mathsf{translate}(m_2)$$
$$\mathsf{translate}(c) = c$$

Fig. 7. Simple algorithm.

and returns an ordered set θ and an annotated form of e. $\mathscr{PL}$ corresponds to the rules in figure 5. Notice that types in figure 5 do not play a role in computing annotations. So, if we assume the input term is well-typed, the algorithm can safely ignore types altogether. We also assume that all variables in the input term are distinct.

In the third rule, the variable is a function name and so should not be included in the returned set. This guarantees that function names will not be lifted.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

$$\mathsf{solve}(\{\theta_{g_i} = \theta_{g_j} \cup \theta\} \cup \{\theta_{g_j} = \theta_j\} \cup \Phi) =$$
$$\quad \mathsf{solve}(\{\theta_{g_i} = \theta_j \backslash_{\theta_{g_i}} \cup \theta\} \cup \{\theta_{g_j} = \theta_j\} \cup \Phi)$$

$\mathsf{solve}(\Phi) = \Phi$, if no θ_{g_i} appears on the RHS of a constraint in Φ.

Fig. 8. Constraint solver.

In the letrec rule, the context Δ' must include a mapping for each function name declared in the letrec to handle mutual recursion. Since the sets associated with these function names have not been computed at this point, the set variables θ_{g_i} are used instead.

Each θ_i returned by a recursive call to $\mathscr{PL}$ represents the variables to be lifted from the function and may include occurrences of set variables. At this point, the algorithm has enough information to compute the actual sets of lifted parameters. A constraint is generated for each function g_i declared in the letrec equating the set variable θ_{g_i} with the set θ_i and passed to a constraint solver. Note that it is safe, and in fact necessary, to remove any occurrence of θ_{g_i} in θ_i (represented by the operation $\theta_i \backslash_{\theta_{g_i}}$).

The constraint solver, defined in figure 8, returns a substitution mapping each θ_{g_i} to a set of variables satisfying the constraints. The first rule eliminates a set variable on the right hand side of a constraint, replacing it with the appropriate set thus far computed. The second rule returns the set of equalities, which serves as a substitution, once all set variables (except those introduced by an enclosing letrec) have been removed from the RHS of all constraints.

The function translate is straightforward. The syntax $y :: s$ represents an ordered set where y is the first element in the set and s is the remainder of the set.

To prove this algorithm correct, we need to prove that it is sound with respect to the two-phase specification of the previous subsection. Again, we need to define a correspondence between different types of contexts:

Definition 5

Let Γ be a context mapping variables to types and Δ be a context mapping variables to sets of variables, such that $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Delta)$. Then $\Gamma \star \Delta$ is the context such that, for all $v \in \mathsf{dom}(\Gamma)$:

$$(\Gamma \star \Delta)(v) = (\Gamma(v), \Delta(v))$$

We can easily extend the notion of closed contexts to Δ:

Definition 6

A context Δ is *closed*, written $\mathsf{Closed}(\Delta)$, iff for all $v \in \mathsf{dom}(\Delta)$,

1. if $\Delta(f) = \theta$ then $\theta \subseteq \mathsf{dom}(\Delta)$, and
2. if $\Delta(x) = \theta$ then $\theta = \emptyset$.

Note that, for any Γ where $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Delta)$, if $\mathsf{Closed}(\Delta)$, then $\mathsf{Closed}(\Gamma \star \Delta)$.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

We must also demonstrate the correctness of the constraint solver. The following results are required by the proof of the soundness theorem below.

Lemma 1
For any finite set of constraints $\Phi = \{\overline{\theta_{g_i} = \theta_i \backslash \theta_{g_i}}\}$:

1. solve(Φ) halts, and
2. if solve(Φ) $= \varepsilon$, then $\varepsilon\theta_{g_i} = \varepsilon\theta_i \backslash \theta_{g_i}$ for all θ_{g_i} in Φ.

The proofs of both parts can be found in Appendix A.

We are now prepared to state the soundness of the parameter lifting algorithm.

Theorem 6 (*Algorithm Soundness*)

1. If Closed(Δ), $\mathscr{PL}(\Delta, e) = (\theta, m)$, and $\Gamma \rhd e : \tau$ then $(\Gamma \star \Delta) \rhd e : (\tau, \theta) \Rightarrow m$.
2. If translate(m) $= e$ then $m \Rightarrow_t e$.

The proof can be found in Appendix A.

Theorem 7 (*Algorithm Correctness*)
If for some well-typed, closed term e, $\mathscr{PL}(\cdot, e) = (\emptyset, m)$ and *translate*(m) $= e'$ then $e \hookrightarrow v$ iff $e' \hookrightarrow v$.

The proof follows immediately from Theorem 6, Theorem 5, and Corollary 1.

The computation of the set θ guarantees that *all* variables (except function names) are lifted from every named function. Parameter lifting can then be followed by a block floating transformation that lifts all named functions to the global level. If we restrict our language such that all functions are named (i.e. only defined using letrec), then the parameter lifting algorithm guarantees that the only free variables occurring inside a function body are function names, which are all global.

This algorithm is limited in the fact that function names are applied directly to lifted variables. Because of this limitation and the simplicity of the source language, types are essentially ignored. As noted in previous sections, our specification supports other possible placements of these applications. In these cases, simply mapping the function name to θ in the context is insufficient. Instead, the set of lifted variables must be included in the function's type. In previous work, we studied type systems for specifying closure conversion (Hannan, 1995), an escape analysis (Hannan, 1998), and a live-variable analysis (Hannan *et al.*, 1997). In each of these, we use types annotated with sets of variables corresponding to the variables needed by a function. We can adapt these specifications to capture the set of variables we need to lift from function definitions.

In the case of higher-order functions, judicious placement of applications can avoid the introduction of new function calls. To ensure that lifting inserts no new function-call sites requires, at least, that parameter lifting be intertwined with block floating to avoid lifting parameters outside of their scope.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

6 Dependent types for parameter lifting

Consider the following program fragment which contains v and w free:

```
letrec f = λx.x+v;
       g = λy.y*w;
       h = λk.k @ 5
in (h @ f) + (h @ g)
```

This expression can be parameter lifted via Johnsson-style algorithms to

```
letrec f = λv.λx.x+v
       g = λw.λy.y*w
       h = λk.k 5
in (h @ (f @ v)) + (h @ (g @ w))
```

in which f and g are partially applied to their arguments, and hence cannot be unCurried.

As already suggested, our specification supports a higher-order form of parameter lifting in which the names of parameter lifted functions can still be passed as arguments, allowing for unCurrying:

```
letrec f = λv.λw.λx.x+v
    g = λv.λw.λy.y*w
    h = λv.λw.λk.k @ v @ w @ 5
in (h @ v @ w @ f) + (h @ v @ w @ g)
```

In this example we must lift parameters v and w from both f and g since both functions occur as the third argument to h. This is a kind of parameter lifting not supported by Johnsson-style algorithms. (Johnsson-style algorithms can lift out parameters not occurring free in a function but only when these parameters are needed by functions occurring in some call chain in which this function occurs.) Observe that the type of both f and g, as determined by our specification, is

$$\{v\}_{\mathsf{int}} \to \{w\}_{\mathsf{int}} \to \mathsf{int} \to \mathsf{int}.$$

Another possibility exists for parameter lifting which still supports the unCurrying of functions f and g:

```
letrec f = λv.λx.x+v
    g = λw.λy.y*w
    h = λa.λk.k @ a @ 5
in (h @ v @ f) + (h @ w @ g)
```

This version exploits the fact that f and g each have one, though not identical, lifted parameter. Our specification of parameter lifting does not support the translation

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

of the original program to this one. We cannot give the same type to `f` and `g`, which is required for them both to occur as the second argument to `h`.

To understand how to support this translation, consider the required types for the two occurrences of `h`. The first occurrence must have type

$$\mathsf{int} \to (\{\mathrm{v}\}_{\mathsf{int}} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int}$$

while the second occurrence must have type

$$\mathsf{int} \to (\{\mathrm{w}\}_{\mathsf{int}} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int}.$$

Observe that the type of the second argument to `h` *depends* on the first argument to `h`. This suggests *dependent types* which, in fact, provide a solution. We can enrich our type system with dependent types as follows

$$\tau \quad ::= \quad \cdots \mid \Pi x{:}\tau.\tau$$

(A dependent type $\Pi x{:}\tau_1.\tau_2$ denotes a function with formal parameter $x : \tau_1$ and whose result type depends of the actual parameter $v : \tau_1$ supplied at each function call. If f has type $\Pi x{:}\tau_1.\tau_2$ and v has type τ_1 then $(f \ @ \ v)$ has type $\tau_2[v/x]$.) Dependent types extend the notion of function types when expression variables can appear free in types. (If x does not occur free in τ_2 then $\Pi x{:}\tau_1.\tau_2$ is equivalent to $\tau_1 \to \tau_2$.) Since expression variables can appear free in our parameter lifting types, our use of dependent types is non-trivial.

We introduce the following rules which are adapted from the rules (lift-app) and (lift-abs) to use dependent types:

$$\frac{\Gamma \rhd e : \Pi x{:}\tau_1.\tau \Rightarrow e' \quad \Gamma(y) = \tau_1}{\Gamma \rhd e : \tau[y/x] \Rightarrow e' \ @ \ y} \ (\Pi\text{-lift-app})$$

$$\frac{x \notin \mathsf{dom}(\Gamma) \quad \Gamma\{x{:}\tau_1\} \rhd \lambda y.e : \tau \Rightarrow \lambda z.e'}{\Gamma \rhd \lambda y.e : \Pi x{:}\tau_1.\tau \Rightarrow \lambda x.\lambda z.e'} \ (\Pi\text{-lift-abs})$$

Adding these rules to our system now allows us to translate the original program to the third translation above. The function `h` can be given type

$$\Pi a{:}\mathsf{int}.(\{\mathrm{a}\}_{\mathsf{int}} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int}$$

and the expressions `(h @ v @ f)` and `(h @ w @ g)` can each be typed accordingly.

Both the second and third translations above support the unCurrying of lifted functions and do not increase the number of function applications at run time. Which of the two, then, should be preferred by an implementation? At first glance, the third requires one less parameter to the lifted form of function `h`, and so this case might be preferred. However, if we assume parameters are passed in registers if possible, then the second case actually might be preferable. If we assume that function `h` expects its three parameters in registers r_0, r_1, and r_2, then we need only copy `v` and `w` once into r_0 and r_1, respectively. Then each call to `h` need only copy

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

the appropriate argument into r_3 (first `f`, then `g`). For this particular example, the difference between the two approaches is insignificant since both require exactly one copying of `v` and `w`.

7 Lambda lifting and polymorphism

We can extend our system to handle polymorphic functions. A polymorphic function can have any free variable lifted from it without affecting the polymorphic nature of the function. We must modify the rule for letrec in figure 3 as follows:

$$\frac{\begin{array}{c} f_i \notin \mathsf{dom}(\Gamma) \\ FV(\tau) \subseteq \mathsf{dom}(\Gamma) \\ \Gamma\{f_i{:}\forall\vec{\alpha}_i.\tau_i\} \rhd e_i : \tau_i \Rightarrow e_i' \qquad \vec{\alpha}_i = FTV(\tau_i) - FTV(\Gamma) \qquad \Gamma\{f{:}\forall\vec{\alpha}_i.\tau_i\} \rhd e : \tau \Rightarrow e' \end{array}}{\Gamma \rhd \mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ e : \tau \Rightarrow \mathsf{letrec}\ \overline{f_i = e_i'}\ \ \mathsf{in}\ e'}$$

When generalizing the type τ, we must ensure that any universally quantified type variable α occurs free in τ but does not occur free in Γ. We use $FTV(\tau)$ to denote the set of free type variables in τ. Likewise for a context Γ.

The treatment of polymorphism requires no special consideration of parameter lifting. To understand why this is so we consider the constraints on the generalization of type variables without the presence of lambda lifting. Any type variable in τ_1 that can be generalized cannot occur free in the context Γ. Any lambda-lifted parameter y in e_1 occurs free in Γ. (This constraint is imposed by the rule (lift-abs).) Hence, any type variable occurring in the type of y (and in τ_1) cannot be generalized. As stated previously, the occurrence of specific variable names in lambda lifting is significant, and this example of handling polymorphism is yet another instance of this observation.

The only additional restriction we need to enforce is that function variables with polymorphic types cannot be lifted. This restriction follows the approach of Johnsson (1985). The restriction of lifting only variables with simple types is inherent in our type-based approach. Because a lifted parameter occurs as the operand in an application, it can assume only a single type. Consider the following example:

```
letrec id = λx.x
in  letrec sqr = λy.y*y
    in letrec f = λg.λz. (id @ g) @ (id @ (z+1))
    in  f @ sqr @ 0
```

The identifier `id` occurs free in the definition of `f` and so we would be tempted to lift it out:

```
letrec id = λx.x
in  letrec sqr = λy.y*y
    in letrec f = λid.λg.λz. (id @ g) @ (id @ (z+1))
    in  f @ id @ sqr @ 0
```

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

Unfortunately, this resulting program is no longer well-typed. The definition of `f` cannot be typed because the parameter `id` cannot be given a simple type, as required by the rule for λ-abstraction.

This restriction to lifting only variables with simple types illustrates a difference between our type-based approach and the (non-typed based) approaches of previous work (Johnsson, 1985). Previous approaches used transformations that introduced intermediate terms that might not be well-typed by the source languages' type systems. This observation has previously been made by Peter Thiemann (1999). Our restriction to well-typed terms, while prohibiting us from lifting out variables of polymorphic type, still allows us to transform a program into a set of global function definitions without any local definitions. If all function names are globally defined, then each function can use any other function. The free variables of a function will consist only of other function names. So while non-type-based approaches can generate supercombinators (functions containing no free variables), we are restricted to fully λ-lifted functions (functions whose free variables are restricted to global function names). Thiemann suggests lifting out a parameter for each different type instance at which the variable is used. An alternative is to consider a richer type system that supports first-class polymorphism.

8 Conclusion

We have presented a declarative specification for lambda lifting and proven it correct with respect to an operational semantics. The specification provides a general relation between a term and a lifted form of the term, without enforcing a single lifting strategy. The symmetry between lifting and dropping is evident from the relational nature of the specification. Thus, this specification provides a foundation from which existing algorithms may be proved correct, and also a starting point for the development of new, type-inference-based, algorithms.

A Selected Proofs

Proof of Theorem 2

The proof of each part follows by induction over the deduction Ξ of $\Gamma \rhd e : \tau \Rightarrow e'$. We consider both parts simultaneously.

1. Ξ is

$$\frac{\Gamma(x) = \tau}{\Gamma \rhd x : \tau \Rightarrow x}\ (\text{var}).$$

From $\Gamma(x) = \tau$ we have $|\Gamma|(x) = |\tau|$ and $||\Gamma||(x) = ||\tau||$. Hence we can construct

$$\frac{|\Gamma|(x) = |\tau|}{|\Gamma| \rhd x : |\tau|} \quad \text{and} \quad \frac{||\Gamma||(x) = ||\tau||}{||\Gamma|| \rhd x : ||\tau||}.$$

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

2. Ξ is

$$\frac{\Sigma(c) = \tau}{\Gamma \rhd c : \tau \Rightarrow c}\ (\text{const}).$$

Because we assume Σ maps constants to unannotated types, $|\tau| = ||\tau|| = \tau$. Then trivially we have

$$\frac{|\Sigma|(c) = |\tau|}{|\Gamma| \rhd c : |\tau|} \quad \text{and} \quad \frac{||\Sigma||(c) = ||\tau||}{||\Gamma|| \rhd c : ||\tau||}.$$

3. Ξ is of the form

$$\frac{\begin{array}{c}\Xi_1 \\ \Gamma\{y{:}\tau\} \rhd e : \tau_1 \Rightarrow e' \quad y \notin \mathsf{dom}(\Gamma) \quad FV(\tau \to \tau_1) \subseteq \mathsf{dom}(\Gamma)\end{array}}{\Gamma \rhd \lambda y.e : \tau \to \tau_1 \Rightarrow \lambda y.e'}\ (\text{abs}).$$

By induction on Ξ_1 we can construct

$$\begin{array}{rcl} \Xi_1' & :: & |\Gamma\{y{:}\tau\}| \rhd e : |\tau_1| \\ \Xi_1'' & :: & ||\Gamma\{y{:}\tau\}|| \rhd e' : ||\tau_1|| \end{array}$$

Observe that $|\Gamma\{y{:}\tau\}| = |\Gamma|\{y{:}|\tau|\}$ and $||\Gamma\{y{:}\tau\}|| = ||\Gamma||\{y{:}||\tau||\}$. Hence we can construct the deductions

$$\frac{\begin{array}{c}\Xi_1' \\ |\Gamma|\{y{:}|\tau|\} \rhd e : |\tau_1|\end{array}}{|\Gamma| \rhd \lambda y.e : |\tau| \to |\tau_1|}$$

and

$$\frac{\begin{array}{c}\Xi_1'' \\ ||\Gamma||\{y{:}||\tau||\} \rhd e' : ||\tau_1||\end{array}}{||\Gamma|| \rhd \lambda y.e' : ||\tau|| \to ||\tau_1||},$$

which are the required deductions because $|\tau_1| \to |\tau| = |\tau_1 \to \tau|$ and $||\tau_1|| \to ||\tau|| = ||\tau_1 \to \tau||$.

4. Ξ is of the form

$$\frac{\begin{array}{c}\Xi_1 \\ \Gamma(x) = \tau_1 \quad \Gamma \rhd \lambda y.e : \tau \Rightarrow \lambda z.e'\end{array}}{\Gamma \rhd \lambda y.e : \{x\}_{\tau_1} \to \tau \Rightarrow \lambda x.\lambda z.e'}\ (\text{lift-abs}).$$

By induction on Ξ_1 we can construct

$$\begin{array}{rcl} \Xi' & :: & |\Gamma| \rhd \lambda y.e : |\tau| \\ \Xi_1'' & :: & ||\Gamma|| \rhd \lambda z.e' : ||\tau|| \end{array}$$

Since $|\{x\}_{\tau_1} \to \tau| = |\tau|$, Ξ' is the required deduction of

$$|\Gamma| \rhd \lambda y.e : |\{x\}_{\tau_1} \to \tau|.$$

Because $\Gamma(x) = \tau_1$, $||\Gamma||(x) = ||\tau_1||$ and hence from Ξ_1'' we can construct

$$\Xi_1' \quad :: \quad ||\Gamma||\{x{:}||\tau_1||\} \rhd \lambda z.e' : ||\tau||$$

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

from which we can construct

$$\frac{\begin{array}{c}\Xi_1'\\ ||\Gamma||\{x{:}||\tau_1||\} \rhd \lambda z.e' : ||\tau||\end{array}}{||\Gamma|| \rhd \lambda x.\lambda z.e' : ||\tau_1|| \to ||\tau||}$$

which is the required deduction because $||\tau_1|| \to ||\tau|| = ||\tau_1 \to \tau||$.

5. Ξ is of the form

$$\frac{\begin{array}{c}\Xi_1\\ \Gamma \rhd e_1 : \tau_2 \to \tau \Rightarrow e_1'\end{array} \quad \begin{array}{c}\Xi_2\\ \Gamma \rhd e_2 : \tau_2 \Rightarrow e_2'\end{array}}{\Gamma \rhd e_1 \mathbin{@} e_2 : \tau \Rightarrow e_1' \mathbin{@} e_2'} \text{ (app)}$$

By induction on Ξ_1 and Ξ_2 we have

$$\begin{array}{lcl}\Xi_1' & :: & |\Gamma| \rhd e_1 : |\tau_2 \to \tau|\\ \Xi_1'' & :: & ||\Gamma|| \rhd e_1' : ||\tau_2 \to \tau||\\ \Xi_2' & :: & |\Gamma| \rhd e_2 : |\tau_2|\\ \Xi_2'' & :: & ||\Gamma|| \rhd e_2' : ||\tau_2||\end{array}$$

Again noting that $|\tau_2 \to \tau| = |\tau_2| \to |\tau|$ and $||\tau_2 \to \tau|| = ||\tau_2|| \to ||\tau||$ we can construct the deductions

$$\frac{\begin{array}{c}\Xi_1'\\ |\Gamma| \rhd e_1 : |\tau_2| \to |\tau|\end{array} \quad \begin{array}{c}\Xi_2'\\ |\Gamma| \rhd e_2 : |\tau_2|\end{array}}{|\Gamma| \rhd e_1 \mathbin{@} e_2 : |\tau|}$$

and

$$\frac{\begin{array}{c}\Xi_1''\\ ||\Gamma|| \rhd e_1' : ||\tau_2|| \to ||\tau||\end{array} \quad \begin{array}{c}\Xi_2''\\ ||\Gamma|| \rhd e_2' : ||\tau_2||\end{array}}{||\Gamma|| \rhd e_1' \mathbin{@} e_2' : ||\tau||}.$$

6. Ξ is of the form

$$\frac{\begin{array}{c}\Xi_1\\ \Gamma \rhd e : \{x\}_{\tau_1} \to \tau \Rightarrow e'\end{array} \quad \Gamma(x) = \tau_1}{\Gamma \rhd e : \tau \Rightarrow e' \mathbin{@} x} \text{ (lift-app)}$$

By induction on Ξ_1, we can construct

$$\begin{array}{lcl}\Xi_1' & :: & |\Gamma| \rhd e : |\{x\}_{\tau_1} \to \tau|\\ \Xi_1'' & :: & ||\Gamma|| \rhd e' : ||\{x\}_{\tau_1} \to \tau||\end{array}$$

Because $|\{x\}_{\tau_1} \to \tau| = |\tau|$, Ξ_1' is the first required deduction.
From $\Gamma(x) = \tau_1$, we have $||\Gamma||(x) = ||\tau_1||$. Observe that $||\{x\}_{\tau_1} \to \tau|| = ||\tau_1|| \to ||\tau||$. From these facts and Ξ_1'' we can construct

$$\frac{\begin{array}{c}\Xi_1''\\ ||\Gamma|| \rhd e' : ||\tau_1|| \to ||\tau||\end{array} \quad \dfrac{||\Gamma||(x) = ||\tau_1||}{||\Gamma|| \rhd x : ||\tau_1||}}{||\Gamma|| \rhd e' \mathbin{@} x : ||\tau||}.$$

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

7. Ξ is of the form

$$\frac{\Xi_j \quad f_i \notin \mathsf{dom}(\Gamma) \quad \Xi_0}{\Gamma\{\overline{f_i{:}\tau_i}\} \rhd e_j : \tau_j \Rightarrow e'_j \quad FV(\tau) \subseteq \mathsf{dom}(\Gamma) \quad \Gamma\{\overline{f_i{:}\tau_i}\} \rhd e : \tau \Rightarrow e'} \text{ (letrec)}$$
$$\Gamma \rhd \mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ e : \tau \Rightarrow \mathsf{letrec}\ \overline{f_i = e'_i}\ \ \mathsf{in}\ e'$$

By induction on the Ξ_j and Ξ_0 we have (for $j = 1$ to n)

$$\begin{aligned}
\Xi'_j &:: |\Gamma\{\overline{f_i{:}\tau_i}\}| \rhd e_j : |\tau_j| \\
\Xi'_0 &:: |\Gamma\{\overline{f_i{:}\tau_i}\}| \rhd e : |\tau| \\
\Xi''_j &:: ||\Gamma\{\overline{f_i{:}\tau_i}\}|| \rhd e'_j : ||\tau_j|| \\
\Xi''_0 &:: ||\Gamma\{\overline{f_i{:}\tau_i}\}|| \rhd e' : ||\tau||
\end{aligned}$$

Hence we can construct the deductions

$$\frac{\Xi'_j \quad \Xi'_0}{|\Gamma|\{\overline{f_i{:}|\tau_i|}\} \rhd e_j : |\tau_j| \quad |\Gamma|\{\overline{f_i{:}|\tau_i|}\} \rhd e : |\tau|}$$
$$|\Gamma| \rhd \mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ e : |\tau|$$

and

$$\frac{\Xi''_j \quad \Xi''_0}{||\Gamma||\{\overline{f_i{:}||\tau_i||}\} \rhd e'_j : ||\tau_j|| \quad ||\Gamma||\{\overline{f_i{:}||\tau_i||}\} \rhd e' : ||\tau||}$$
$$||\Gamma|| \rhd \mathsf{letrec}\ \overline{f_i = e'_i}\ \ \mathsf{in}\ e' : ||\tau|| \qquad .$$

□

Proof of Theorem 4

We give the proof to part 1. It follows by induction on the pair $(|\Pi|, |\Xi|)$ where $\Pi :: \rhd e \hookrightarrow v$ and $\Xi :: \rhd e : \tau \Rightarrow e'$. We show how to construct $\Pi' :: \rhd e' \hookrightarrow v'$ and $\Xi' :: \rhd v : \tau \Rightarrow v'$.

The proof makes use of some properties of deductions that allows us to construct new deductions from existing ones. In particular, given a deduction of $\Xi :: (x : \tau_1 \Rightarrow x' \supset e : \tau \Rightarrow e')$ in which x and x' are variables and a deduction $\Xi_1 :: e_1 : \tau_1 \Rightarrow e'_1$, by substitution (function application in constructive type theory) we can construct a deduction $\Xi' :: e[e_1/x] : \tau \Rightarrow e'[e'_1/x']$. This result can be justified when we view deductions as objects and interpret logical implication as the function type constructor. Our previous work on unCurrying (Hannan & Hicks, 2000) makes significant use of this and provides further explanation of the technique. (If we had used explicit type contexts we would have required a substitution lemma for this kind of result.)

1. Assume Π is $\overline{\lambda x.e \hookrightarrow \lambda x.e}$ and Ξ is a deduction of $\lambda x.e : \tau \Rightarrow \lambda z.e'$. Then Π' is $\overline{\lambda z.e' \hookrightarrow \lambda z.e'}$ and $\Xi' = \Xi$.
2. Assume Π is

$$\frac{\Pi_1 \quad \Pi_2}{\rhd e_1 \hookrightarrow \lambda x.e \quad \rhd e[e_2/x] \hookrightarrow v}$$
$$\rhd e_1 \mathbin{@} e_2 \hookrightarrow v$$

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

and Ξ is

$$\frac{\begin{array}{cc}\Xi_1 & \Xi_2\\ \rhd e_1 : (\tau_2 \to \tau_1) \Rightarrow e_1' & \rhd e_2 : \tau_2 \Rightarrow e_2'\end{array}}{\rhd e_1 \mathbin{@} e_2 : \tau_1 \Rightarrow e_1' \mathbin{@} e_2'}$$

By induction on Π_1 and Ξ_1 there exists a v_1' such that

$$\Pi_1' \quad :: \quad e_1' \hookrightarrow v_1' \tag{A 1}$$

$$\Xi_1' \quad :: \quad \lambda x.e : (\tau_2 \to \tau_1) \Rightarrow v_1' \tag{A 2}$$

By the structure of the inference rules, then Ξ_1' must be of the form

$$\frac{\begin{array}{c}\Xi_1''\\ \forall c, c'(c : \tau_2 \Rightarrow c' \ \supset \ e[c/x] : \tau_1 \Rightarrow e'[c'/x'])\end{array}}{\lambda x.e : (\tau_2 \to \tau_1) \Rightarrow \lambda x'.e'}$$

where $v_1' = \lambda x'.e'$ for some x' and e'. We can instantiate Ξ_1'' (using e_2, e_2', and Ξ_2) yielding a deduction

$$\Xi_3 \quad :: \quad e[e_2/x] : \tau_1 \Rightarrow e'[e_2'/x'] \tag{A 3}$$

By induction on Π_2 and Ξ_3, there exists a v' such that

$$\Pi_2' \quad :: \quad e'[e_2'/x'] \hookrightarrow v' \tag{A 4}$$

$$\Xi' \quad :: \quad v : \tau_1 \Rightarrow v' \tag{A 5}$$

Finally, we can construct the deduction Π':

$$\frac{\begin{array}{cc}\Pi_1' & \Pi_2'\\ \rhd e_1' \hookrightarrow \lambda x'.e' & \rhd e'[e_2'/x'] \hookrightarrow v'\end{array}}{\rhd e_1' \mathbin{@} e_2' \hookrightarrow v'}$$

3. Assume Π is an arbitrary deduction of $\rhd e \hookrightarrow v$ and Ξ is of the form

$$\frac{\begin{array}{cc}\Xi_1 & \Xi_2\\ \rhd e : \{w\}_{\tau_1} \to \tau \Rightarrow e' & \rhd v_1 : \tau_1 \Rightarrow w\end{array}}{\rhd e : \tau \Rightarrow e' \mathbin{@} w}$$

Then by induction on Π and Ξ_1 there exists a v' such that

$$\Pi_1' \quad :: \quad e' \hookrightarrow v' \tag{A 6}$$

$$\Xi_1' \quad :: \quad v : \{w\}_{\tau_1} \to \tau \Rightarrow v' \tag{A 7}$$

The deduction Ξ_1' must be of the form

$$\frac{\begin{array}{ccc}\Xi' & \Xi_2' & \Xi_3'\\ \lambda y.e_1 : \tau \Rightarrow \lambda z.e_1'[w/x'] & x : \tau_1 \Rightarrow x' & v_1 : \tau_1 \Rightarrow w\end{array}}{\lambda y.e_1 : \{w\}_{\tau_1} \to \tau \Rightarrow \lambda x'.\lambda z.e_1'}$$

in which $v = \lambda y.e_1$ for some y and e_1, and $v' = \lambda x'.\lambda z.e_1'$ for some z and e_1'. Then Π' can be constructed as

$$\frac{\begin{array}{cc}\Pi_1' & \overline{\lambda z.e_1'[w/x'] \hookrightarrow \lambda z.e_1'[w/x']}\end{array}}{e' \mathbin{@} w \hookrightarrow \lambda z.e_1'[w/x']}$$

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

4. Assume Π is

$$\dfrac{\dfrac{\Pi_0}{e[(\mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ \ e_j)/f_j] \hookrightarrow v}}{\mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ \ e \hookrightarrow v}$$

(in which $1 \leqslant i \leqslant n$ for some $n \geqslant 1$) and Ξ is

$$\dfrac{\Xi_1 \quad \cdots \quad \Xi_n \quad \Xi_0}{\mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ \ e : \tau \Rightarrow \mathsf{letrec}\ \overline{f'_i = e'_i}\ \ \mathsf{in}\ \ e'}$$

in which

$$\Xi_1 \quad ::= \quad \forall \overline{c_i}, \overline{c'_i}(\overline{c_i : \tau_i \Rightarrow c'_i} \supset e_1[\overline{c_i/f_i}] : \tau_1 \Rightarrow e'_1[\overline{c'_i/f'_i}]) \tag{A 8}$$

$$\vdots \qquad\quad \vdots$$

$$\Xi_n \quad ::= \quad \forall \overline{c_i}, \overline{c'_i}(\overline{c_i : \tau_i \Rightarrow c'_i} \supset e_n[\overline{c_i/f_i}] : \tau_n \Rightarrow e'_1[\overline{c'_i/f'_i}]) \tag{A 9}$$

$$\Xi_0 \quad ::= \quad \forall \overline{c_i}, \overline{c'_i}(\overline{c_i : \tau_i \Rightarrow c'_i} \supset e[\overline{c_i/f_i}] : \tau \Rightarrow e'[\overline{c'_i/f'_i}]) \tag{A 10}$$

Observe that for $1 \leqslant k \leqslant n$ we can construct a deduction Ξ'_k of

$$\dfrac{\Xi_1 \quad \cdots \quad \Xi_n \quad \Xi_k}{\mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ \ e_k : \tau_k \Rightarrow \mathsf{letrec}\ \overline{f'_i = e'_i}\ \ \mathsf{in}\ \ e'_k}$$

Applying Ξ_0 to the deductions $\Xi'_1, \ldots, \Xi'_n$ we obtain a deduction

$$\Xi'_0 \quad :: \quad e[\overline{(\mathsf{letrec}\ \overline{f_i = e_i}\ \ \mathsf{in}\ \ e_j)/f_j}] : \tau \Rightarrow e'[\overline{(\mathsf{letrec}\ \ \overline{f'_i = e'_i}\ \ \mathsf{in}\ \ e'_j)/f'_j}] \tag{A 11}$$

By induction on Π_0 and Ξ'_0 we have that there exists a v' such that

$$\Pi'_0 \quad :: \quad e'[\overline{(\mathsf{letrec}\ \overline{f'_i = e'_i}\ \ \mathsf{in}\ \ e'_j)/f'_j}] \hookrightarrow v' \tag{A 12}$$

$$\Xi' \quad :: \quad v : \tau \Rightarrow v' \tag{A 13}$$

Hence we can construct Π' as

$$\dfrac{\dfrac{\Pi'_0}{e'[\overline{(\mathsf{letrec}\ \overline{f'_i = e'_i}\ \ \mathsf{in}\ \ e'_j)/f'_j}] \hookrightarrow v'}}{\mathsf{letrec}\ \overline{f'_i = e'_i}\ \ \mathsf{in}\ \ e' \hookrightarrow v'}$$

Part 2 follows similarly. □

Proof of Theorem 5

Before proceeding with the proof of the theorem we introduce two useful auxiliary lemmas.

Lemma 2

If $\Lambda \cong \Gamma$ then $FV(\Lambda) = FV(\Gamma)$.

The proof is straightforward from Definitions 2 and 3.

Lemma 3

If $\mathsf{Closed}(\Lambda)$ and $\Lambda \rhd e : (\tau, \theta) \Rightarrow m$ then $\theta \subseteq \mathsf{dom}(\Lambda)$.

The proof is straightforward by induction on the typing derivation.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

Now the proof of the theorem proceeds by well-founded induction on the structure of the typing derivation $\Xi :: \Lambda \rhd e : (\tau, \theta) \Rightarrow m$.

1. Ξ is

$$\frac{\Sigma(c) = \tau}{\Lambda \rhd c : (\tau, \emptyset) \Rightarrow c} \text{ (const)}$$

and Δ is

$$\overline{c \Rightarrow_t c}.$$

Then trivially Ξ' is

$$\frac{\Sigma(c) = \tau}{\Gamma \rhd c : \tau \Rightarrow c} \text{ (const)}$$

2. Ξ is

$$\frac{\Lambda(x) = (\tau, \emptyset)}{\Lambda \rhd x : (\tau, \{x\}) \Rightarrow x^{\emptyset}}.$$

Then Δ is $\overline{x^{\emptyset} \Rightarrow_t x}$, and Ξ' is

$$\frac{\Gamma(x) = \tau}{\Gamma \rhd x : \tau \Rightarrow x}.$$

3. Ξ is

$$\frac{\Lambda(f) = (\tau, \theta)}{\Lambda \rhd f : (\tau, \theta) \Rightarrow f^{\theta}}.$$

Suppose θ is $y_1 :: y_2 :: \cdots :: y_n$ for some $n \geqslant 0$. Then Δ is a deduction of

$$f^{\theta} \Rightarrow_t f \mathbin{@} y_n \mathbin{@} \cdots \mathbin{@} y_2 \mathbin{@} y_1$$

and by Definition 3, $\Gamma(f) = \tau'$ such that

$$\tau' = \{y_n\}_{\tau_{y_n}} \to \cdots \{y_2\}_{\tau_{y_2}} \to \{y_1\}_{\tau_{y_1}} \to \tau$$

(where $\Gamma(y_i) = y_i$ for $i \in 1..n$).
Then using n instances of (lift-app) we can construct Ξ' as

$$\dfrac{\dfrac{\dfrac{\Gamma(f) = \{y_n\}_{\tau_{y_n}} \to \cdots \{y_2\}_{\tau_{y_2}} \to \{y_1\}_{\tau_{y_1}} \to \tau}{\Gamma \rhd f : \{y_n\}_{\tau_{y_n}} \to \cdots \{y_2\}_{\tau_{y_2}} \to \{y_1\}_{\tau_{y_1}} \to \tau \Rightarrow f}}{\vdots}}{\dfrac{\Gamma \rhd f : \{y_2\}_{\tau_{y_2}} \to \{y_1\}_{\tau_{y_1}} \to \tau \Rightarrow f \mathbin{@} y_n \mathbin{@} \cdots \mathbin{@} y_3 \quad \Gamma(y_2) = y_2}{\Gamma \rhd f : \{y_1\}_{\tau_{y_1}} \to \tau \Rightarrow f \mathbin{@} y_n \mathbin{@} \cdots \mathbin{@} y_2} \text{ (lift-app)} \quad \Gamma(y_1) = y_1}{\Gamma \rhd f : \tau \Rightarrow f \mathbin{@} y_n \mathbin{@} \cdots \mathbin{@} y_2 \mathbin{@} y_1}$$

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

4. Ξ is

$$\frac{\begin{array}{c}\Xi_1\\ \Lambda\{y{:}(\tau,\emptyset)\} \rhd e : (\tau_1,\theta) \Rightarrow m \quad y \notin \mathsf{dom}(\Lambda)\end{array}}{\Lambda \rhd \lambda y.e : (\tau \to \tau_1, \theta - \{y\}) \Rightarrow \lambda y.m}$$

and Δ is

$$\frac{\begin{array}{c}\Delta_1\\ m \Rightarrow_t e'\end{array}}{\lambda y.m \Rightarrow_t \lambda y.e'.}$$

From $\Lambda \cong \Gamma$ we have $\Lambda\{y{:}(\tau,\emptyset)\} \cong \Gamma\{y : \tau\}$, and from $\mathsf{Closed}(\Lambda)$ we also have $\mathsf{Closed}(\Lambda\{y{:}(\tau,\emptyset)\})$.
By induction on Ξ_1 and Δ_1, we have

$$\Xi_1' :: \Gamma\{y{:}\tau\} \rhd e : \tau_1 \Rightarrow e'$$

and so we can construct Ξ' as

$$\frac{\begin{array}{c}\Xi_1'\\ \Gamma\{y{:}\tau\} \rhd e : \tau_1 \Rightarrow e'\end{array}}{\Gamma \rhd \lambda y.e : \tau \to \tau_1 \Rightarrow \lambda y.e'.}$$

5. Ξ is

$$\frac{\begin{array}{cc}\Xi_1 & \Xi_2\\ \Lambda \rhd e_1 : (\tau_2 \to \tau, \theta_1) \Rightarrow m_1 & \Lambda \rhd e_2 : (\tau_2, \theta_2) \Rightarrow m_2\end{array}}{\Lambda \rhd e_1 \mathbin{@} e_2 : (\tau, \theta_1 \cup \theta_2) \Rightarrow m_1 \mathbin{@} m_2}$$

and Δ is

$$\frac{\begin{array}{cc}\Delta_1 & \Delta_2\\ m_1 \Rightarrow_t e_1' & m_2 \Rightarrow_t e_2'\end{array}}{m_1 \mathbin{@} m_2 \Rightarrow_t e_1' \mathbin{@} e_2'.}$$

By induction on Ξ_1 and Δ_1 we have

$$\Xi_1' :: \Gamma \rhd e_1 : \tau_2 \to \tau \Rightarrow e_1'.$$

By induction on Ξ_2 and Δ_2 we have

$$\Xi_2' :: \Gamma \rhd e_2 : \tau_2 \Rightarrow e_2'.$$

Hence we can build Ξ' as

$$\frac{\begin{array}{cc}\Xi_1' & \Xi_2'\\ \Gamma \rhd e_1 : \tau_2 \to \tau \Rightarrow e_1' & \Gamma \rhd e_2 : \tau_2 \Rightarrow e_2'\end{array}}{\Gamma \rhd e_1 \mathbin{@} e_2 : \tau \Rightarrow e_1' \mathbin{@} e_2'}.$$

6. Ξ is

$$\frac{\begin{array}{cccc}\Xi_i & & & \Xi'\\ \Lambda' \rhd e_i : (\tau_i, \theta_i) \Rightarrow m_i & f_i \notin \mathsf{dom}(\Lambda) & & \Lambda' \rhd e : (\tau, \theta) \Rightarrow m\end{array}}{\Lambda \rhd \mathsf{letrec}\ \overline{f_i = e_i}\ \mathsf{in}\ e : (\tau, \theta \cup \overline{\theta_i}) \Rightarrow \mathsf{letrec}\ \overline{f_i =^{\theta_i} m_i}\ \mathsf{in}\ m}$$

in which $\Lambda' = \Lambda\{\overline{f_i{:}(\tau_i, \theta_i)}\}$.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

Then Δ is

$$\frac{\Delta_i \qquad \Delta'}{f_i =^{\theta_i} m_i \Rightarrow_t f_i = e_i' \quad m \Rightarrow_t e'}{\mathsf{letrec}\ \overline{f_i =^{\theta_i} m_i}\ \mathsf{in}\ m \Rightarrow_t \mathsf{letrec}\ \overline{f_i = e_i'}\ \mathsf{in}\ e'}$$

Assume θ_i is $x_1 :: x_2 :: \cdots :: x_{n_i} :: \emptyset$ for some $n_i \geqslant 0$. Each deduction Δ_i must be of the form

$$\frac{\dfrac{\dfrac{\dfrac{\Delta_i'}{\lambda x_{n_i}.\cdots\lambda x_2.\lambda x_1.m_i \Rightarrow_t e_i'}}{f_i =^{\emptyset} \lambda x_{n_i}.\cdots\lambda x_2.\lambda x_1.m_i \Rightarrow_t f_i = e_i'}}{\vdots}}{f_i =^{x_2 :: \cdots :: x_{n_i} :: \emptyset} \lambda x_1.m_i \Rightarrow_t f_i = e_i'}}{f_i =^{x_1 :: x_2 :: \cdots :: x_{n_i} :: \emptyset} m_i \Rightarrow_t f_i = e_i'}$$

Then by the structure of the translation relation, $e_i' = \lambda x_{n_i}.\cdots\lambda x_2.\lambda x_1.e_i''$ for some e_i'' and the Δ_i' must be of the form

$$\frac{\dfrac{\dfrac{\Delta_i''}{m_i \Rightarrow_t e_i''}}{\vdots}}{\lambda x_{n_i}.\cdots\lambda x_2.\lambda x_1.m_i \Rightarrow_t \lambda x_{n_i}.\cdots\lambda x_2.\lambda x_1.e_i''}$$

Let $\Gamma' = \Gamma\{\overline{f_i : \tau_i'}\}$ where $(\tau_i, \theta_i) \succ_{\Lambda'} \tau_i'$. Then $\Lambda' \cong \Gamma'$. Observe that we have $\mathsf{Closed}(\Lambda)$. By induction on each Ξ_i and Δ_i'', we have

$$\Xi_i' :: \Gamma' \rhd e_i : \tau_i \Rightarrow e_i''.$$

By Lemma 3 and Ξ_i, $\theta_i \subseteq \mathsf{dom}(\Lambda')$, and hence $\mathsf{Closed}(\Lambda')$ and $\theta_i \subseteq \mathsf{dom}(\Gamma')$. Starting with Ξ_i' and applying the (lift-abs) rule n_i times we can construct Ξ_i'':

$$\frac{\Gamma'(x_{n_i}) = \tau_{x_{n_i}} \qquad \dfrac{\dfrac{\Xi_i'}{\Gamma'(x_1) = \tau_{x_1} \quad \Gamma' \rhd e_i : \tau_i \Rightarrow e_i''}}{\Gamma' \rhd e_i : \{x_1\}_{\tau_{x_1}} \to \tau_i \Rightarrow \lambda x_1.e_i''}}{\vdots}}{\Gamma' \rhd e_i : \{x_{n_i}\}_{\tau_{x_{n_i}}} \to \cdots \{x_2\}_{\tau_{x_2}} \to \{x_1\}_{\tau_{x_1}} \to \tau_i \Rightarrow \lambda x_{n_i} \cdots \lambda x_2.\lambda x_1.e_i''}$$

By induction on Ξ' and Δ' we have

$$\Xi'' :: \Gamma' \rhd e : \tau \Rightarrow e'.$$

From $f_i \notin \mathsf{dom}(\Lambda)$ we have $f_i \notin \mathsf{dom}(\Gamma)$. Let

$$\tau_i' = \{x_{n_i}\}_{\tau_{x_{n_i}}} \to \cdots \{x_2\}_{\tau_{x_2}} \to \{x_1\}_{\tau_{x_1}} \to \tau_i.$$

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

We can construct the required deduction as

$$\frac{\overset{\Xi_i''}{\Gamma' \rhd e_i : \tau_i' \Rightarrow \lambda x_{n_i} \cdots \lambda x_2.\lambda x_1.e_i''} \quad f_i \notin \mathsf{dom}(\Gamma') \quad \overset{\Xi''}{\Gamma' \rhd e : \tau \Rightarrow e'}}{\Gamma \rhd \mathsf{letrec}\ \overline{f_i = e_i}\ \mathsf{in}\ e : \tau \Rightarrow \mathsf{letrec}\ \overline{f_i = \lambda x_{n_i} \cdots \lambda x_2.\lambda x_1.e_i''}\ \mathsf{in}\ e'}$$

□

Proof of Lemma 1

1. The first rule of solve

$$\mathsf{solve}(\{\theta_{g_i} = \theta_{g_j} \cup \theta\} \cup \{\theta_{g_j} = \theta_j\} \cup \Phi) = \mathsf{solve}(\{\theta_{g_i} = \theta_j \backslash_{\theta_{g_i}} \cup \theta\} \cup \{\theta_{g_j} = \theta_j\} \cup \Phi)$$

replaces one occurrence of θ_{g_j} on the right hand side of a constraint with θ_j. Since θ_{g_j} cannot occur in θ_j, one occurrence of θ_{g_j} has been removed from Φ. Since there are a finite number, say N, of θ_{g_j}, they can all be eliminated from the RHS of constraints after N applications of rule 1. Likewise for the rest of the θ_{g_i}. When all θ_{g_i} have been removed from the RHS of all constraints, the resulting constraint list is returned as a substitution. Hense solve(Φ) halts.

2. By the second rule of solve, ε must be a set of equalities $\{\overline{\theta_{g_i} = \theta_i'}\}$ where no θ_{g_i} occur in any θ_i'. We show that ε is a solution to Φ.
We do this by demonstrating the correctness of rule 1; specifically, if ε is a solution to

$$\{\theta_{g_i} = \theta_j \backslash_{\theta_{g_i}} \cup \theta\} \cup \{\theta_{g_j} = \theta_j\} \cup \Phi$$

then it is a solution to

$$\{\theta_{g_i} = \theta_{g_j} \cup \theta\} \cup \{\theta_{g_j} = \theta_j\} \cup \Phi.$$

All of the constraints are the same in the two sets except for θ_{g_i}. Since we know $\varepsilon\theta_{g_j} = \varepsilon\theta_j$,

$$\varepsilon\theta_{g_i} = \varepsilon(\theta_{g_j} \backslash_{\theta_{g_i}} \cup \theta)$$

We can add $\varepsilon\theta_{g_i}$ to the RHS allowing us to remove the restriction on θ_{g_j}, thus:

$$\varepsilon\theta_{g_i} = \varepsilon(\theta_{g_j} \cup \theta)$$

Now, if $\mathsf{solve}(\Phi) \Rightarrow \mathsf{solve}(\Phi') \Rightarrow \cdots \Rightarrow \varepsilon$, we know ε is a solution to Φ, hence $\varepsilon\theta_{g_i} = \varepsilon\theta_i \backslash_{\theta_{g_i}}$ for all θ_{g_i} in Φ.
□

Proof of Theorem 6

We only present the proof of Part 1 here. The proof is by induction over the definition of $\mathscr{PL}$ and the deduction Π of $\Gamma \rhd e : \tau$.

1. Π is

$$\frac{\Sigma(c) = \tau}{\Gamma \rhd c : \tau}$$

and $\mathscr{PL}(\Delta, c) = (\emptyset, c)$. We can construct Ξ:

$$\frac{\Sigma(c) = \tau}{(\Gamma \star \Delta) \rhd c : (\tau, \emptyset) \Rightarrow c}$$

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

2. Π is

$$\frac{\Gamma(x) = \tau}{\Gamma \rhd x : \tau}$$

$\Delta(x) = \theta$, then $\mathscr{PL}(\Delta, x) = (\{x\}, x^{\theta})$. Since $\mathsf{Closed}(\Delta)$, $\theta = \emptyset$ and we can construct Ξ:

$$\frac{(\Gamma \star \Delta)(x) = (\tau, \emptyset)}{(\Gamma \star \Delta) \rhd x : (\tau, \{x\}) \Rightarrow x^{\emptyset}}$$

3. Π is

$$\frac{\Gamma(f) = \tau}{\Gamma \rhd f : \tau}$$

$\Delta(f) = \theta$, then $\mathscr{PL}(\Delta, f) = (\theta, f^{\theta})$. We can construct Ξ:

$$\frac{(\Gamma \star \Delta)(f) = (\tau, \theta)}{(\Gamma \star \Delta) \rhd f : (\tau, \theta) \Rightarrow f^{\theta}}$$

4. Π is

$$\frac{\begin{array}{c}\Pi_1 \\ \Gamma\{y : \tau\} \rhd e : \tau_1\end{array}}{\Gamma \rhd \lambda y.e : \tau \to \tau_1}$$

and $\mathscr{PL}(\Delta, \lambda y.e) = (\theta - \{y\}, \lambda y.m)$. By induction on Π_1 and the recursive call to $\mathscr{PL}$ we have

$$\Xi_1 :: (\Gamma\{y : \tau\} \star \Delta\{y : \emptyset\}) \rhd e : (\tau_1, \theta) \Rightarrow m$$

By Definition 5, $\Gamma\{y : \tau\} \star \Delta\{y : \emptyset\} = (\Gamma \star \Delta)\{y : (\tau, \emptyset)\}$. Since all variable names are distinct, the constraints $y \notin \mathsf{dom}(\Gamma)$ and $y \notin \mathsf{dom}(\Delta)$ are trivially satisfied. Thus $y \notin \mathsf{dom}(\Gamma \star \Delta)$ and we can construct Ξ:

$$\frac{\begin{array}{c}\Xi_1 \\ (\Gamma \star \Delta)\{y : (\tau, \emptyset)\} \rhd e : (\tau_1, \theta) \Rightarrow m \quad y \notin \mathsf{dom}(\Gamma \star \Delta)\end{array}}{(\Gamma \star \Delta) \rhd \lambda y.e : (\tau \to \tau_1, \theta - \{y\}) \Rightarrow \lambda y.m}$$

5. Π is

$$\frac{\begin{array}{cc}\Pi_1 & \Pi_2 \\ \Gamma \rhd e_1 : \tau_1 \to \tau & \Gamma \rhd e_2 : \tau_1\end{array}}{\Gamma \rhd e_1 \mathbin{@} e_2 : \tau}$$

and $\mathscr{PL}(\Delta, e_1 \mathbin{@} e_2) = (\theta_1 \cup \theta_2, m_1 \mathbin{@} m_2)$. By induction on Π_1 and Π_2, and the recursive calls to $\mathscr{PL}$, we have

$$\Xi_1 :: (\Gamma \star \Delta) \rhd e_1 : (\tau_1 \to \tau, \theta_1) \Rightarrow m_1$$

$$\Xi_2 :: (\Gamma \star \Delta) \rhd e_2 : (\tau_1, \theta_2) \Rightarrow m_2$$

We can construct Ξ:

$$\frac{\begin{array}{cc}\Xi_1 & \Xi_2 \\ (\Gamma \star \Delta) \rhd e_1 : (\tau_1 \to \tau, \theta_1) \Rightarrow m_1 & (\Gamma \star \Delta) \rhd e_2 : (\tau_1, \theta_2) \Rightarrow m_2\end{array}}{(\Gamma \star \Delta) \rhd e_1 \mathbin{@} e_2 : (\tau, \theta_1 \cup \theta_2) \Rightarrow m_1 \mathbin{@} m_2}$$

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

6. Π is

$$\frac{\Gamma' = \Gamma\{\overline{g_i : \tau_i \to \tau'_i}\} \quad \overset{\Pi_i}{\Gamma' \rhd \lambda y_i.e_i : \tau_i \to \tau'_i} \quad \overset{\Pi'}{\Gamma' \rhd e : \tau}}{\Gamma \rhd \mathsf{letrec}\ \overline{g_i = \lambda y_i.e_i}\ \mathsf{in}\ e : \tau}$$

and $\mathscr{PL}(\Delta, \mathsf{letrec}\ \overline{g_i = \lambda y_i.e_i}\ \mathsf{in}\ e) = (\bigcup \varepsilon\theta_{g_i} \cup \theta, \mathsf{letrec}\ \overline{g_i =^{\varepsilon\theta_{g_i}} \lambda y_i.\varepsilon m_i}\ \mathsf{in}\ m)$. $\Delta' = \Delta\{\overline{g_i : \theta_{g_i}}\}$ and by induction on Π_i and the recursive calls to $\mathscr{PL}$, we have

$$\Xi_i :: (\Gamma' \star \Delta') \rhd \lambda y_i.e_i : (\tau_i \to \tau'_i, \theta_i) \Rightarrow \lambda y_i.m_i$$

$\varepsilon = \mathsf{solve}(\{\overline{\theta_{g_i} = \theta_i \backslash_{\theta_{g_i}}}\})$ and by Lemma 1, $\varepsilon\theta_{g_i} = \varepsilon\theta_i \backslash_{\theta_{g_i}} = \varepsilon\theta_i$. By applying the substitution ε to the deductions Ξ_i we have

$$\Xi'_i :: (\Gamma' \star \varepsilon\Delta') \rhd \lambda y_i.e_i : (\tau_i \to \tau'_i, \varepsilon\theta_i) \Rightarrow \varepsilon\lambda y_i.m_i$$

By induction on Π' and the recursive call to $\mathscr{PL}$, we have

$$\Xi' :: (\Gamma' \star \varepsilon\Delta') \rhd e : (\tau, \theta) \Rightarrow m$$

All variable names are distinct, so $g_i \notin \mathsf{dom}(\Gamma \star \Delta)$. There are no θ_{g_i} in Δ, so by Definition 5

$$(\Gamma' \star \varepsilon\Delta') = (\Gamma \star \Delta)\{\overline{g_i : (\tau_i, \varepsilon\theta_{g_i})}\}$$

Now we can construct:

$$\frac{\overset{\Xi'_i}{(\Gamma' \star \varepsilon\Delta') \rhd \lambda y_i.e_i : (\tau_i \to \tau'_i, \varepsilon\theta_i) \Rightarrow \varepsilon\lambda y_i.m_i} \quad \overset{\Xi'}{(\Gamma' \star \varepsilon\Delta') \rhd e : (\tau, \theta) \Rightarrow m}}{(\Gamma \star \Delta) \rhd \mathsf{letrec}\ \overline{g_i = \lambda y_i.e_i}\ \mathsf{in}\ e : (\tau, \bigcup \varepsilon\theta_{g_i} \cup \theta) \Rightarrow \mathsf{letrec}\ \overline{g_i =^{\varepsilon\theta_{g_i}} \lambda y_i.\varepsilon m_i}\ \mathsf{in}\ m}$$

□

Acknowledgements

We thank Olivier Danvy and the referees for their comments and suggestions.

References

Aspinall, D. (1995) Subtyping with singleton types. In: Pacholski, L. and Tiuryn, J. (eds.) *Computer Science Logic: Lecture Notes in Computer Science 933*, pp. 1–15. Springer-Verlag.

Bondorf, A. and Danvy, O. (1991) Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, **16**(2), 151–195.

Clinger, W. and Hansen, L. T. (1994) Lambda, the ultimate label, or a simple optimizing compiler for scheme. In: Talcott, C., editor, *Proceedings of ACM Conference on LISP and Functional Programming*, pp. 128–139. ACM Press.

Danvy, O. (1998) *An extensional characterization of lambda lifting*. Technical report RS-98-2, BRICS.

Danvy, O. and Schultz, U. P. (2000) Lambda-dropping: transforming recursive equations into programs with block structure. *Theor. Comput. Sci.* **248**(1–2), 243–287.

Fischbach, A. and Hannan, J. (2000) Specification and correctness of lambda lifting. In: Taha, W., editor, *Semantics, Applications and Implementation of Program Generation: Lecture Notes in Computer Science 1924*, pp. 108–128. Springer-Verlag.

https://doi.org/10.1017/S0956796802004604 Published online by Cambridge University Press

Fischbach, A. and Hannan, J. (2001) Type systems and algorithms for useless-variable elimination. In: Danvy, O. and Filinski, A., editors, *Programs as Data Objects: Lecture Notes in Computer Science 2053*, pp. 25–38. Springer-Verlag.

Hannan, J. (1995) Type systems for closure conversions. In: Nielson, H. R. and Solberg, K. L., editors, *Participants' Proceedings of the Workshop on Types for Program Analysis*, pp. 48–62.

Hannan, J. (1998) A type-based escape analysis for functional languages. *J. Functional Program.* **8**(3), 239–273.

Hannan, J. and Hicks, Pk. (1998) Higher-order arity raising. In: Hudak, P. and Queinnec, C., editors, *Proceedings Third International Conference on Functional Programming*, pp. 27–38. ACM Press.

Hannan, J. and Hicks, P. (2000) Higher-order unCurrying. *Higher-order & Symbolic Computation*, **13**(3), 179–216.

Hannan, J., Hicks, P. and Liben-Nowell, D. (1997) *A lifetime analysis for higher-order languages*. Technical report CSE-97-014, Penn State University.

Hughes, J. (1982) Super combinators: A new implementation method for applicative languages. In: Wise, D. S., editor, *Proceedings ACM Symposium on LISP and Functional Programming*, pp. 1–10. ACM Press.

Johnsson, T. (1985) Lambda lifting: Transforming programs to recursive equations. In: Jouannaud, J.-P., editor, *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 201*, pp. 190–203. Springer-Verlag.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall.

Peyton Jones, S. L. and Lester, D. R. (1992) *Implementing Functional Languages*. Prentice Hall International Series in Computer Science. Prentice Halll.

Peyton Jones, S. L., Partain, W. and Santos, A. (1996) Let-floating: Moving bindings to give faster programs. In: Harper, R., editor, *Proceedings ACM SIGPLAN International Conference on Functional Languages*, pp. 1–12. ACM Press.

Shivers, O. (1991) *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie-Mellon University.

Thiemann, P. (1999) ML-style typing, lambda lifting, and partial evaluation. *Proceedings Latin American Conference on Functional Programming*.