Design and Correctness of Program Transformations Based on Control-Flow Analysis

Anindya Banerjee[1], Nevin Heintze[2], and Jon G. Riecke[3]

[1] Kansas State University, Manhattan, KS 66506, USA,
banerjee@cis.ksu.edu
[2] Agere Systems, Murray Hill, NJ 07974, USA
nch@agere.com
[3] Aleri, Inc., New York, NY 10011, USA
Jon.Riecke@aleri.com

Abstract. We show how control-flow-based program transformations in functional languages can be proven correct. The method relies upon "defunctionalization," a mapping from a higher-order language to a first-order language. We first show that defunctionalization is correct; using this proof and common semantic techniques, we then show how two program transformations—flow-based inlining and lightweight defunctionalization—can be proven correct.

1 Introduction

Control-flow analysis (CFA) [20, 37, 38] is a program analysis that computes an approximation to the set of functions or methods that may be called at a call site. CFA's usefulness stems from the style of programming in functional and object-oriented languages. Typically, programs in such languages have small basic blocks and use recursive functions for loops. These factors limit the applicability of standard intraprocedural optimizations [2, 25], and increase the importance of interprocedural optimizations. CFA enables interprocedural optimizations, and so is used frequently [7, 9, 10, 17, 19, 38, 41, 46].

The correctness of CFA itself has been widely studied [15, 18–20, 28–30, 37, 38], but little is known about how to prove the correctness of CFA-based *transformations*. An analogous situation for dataflow analysis and dataflow-analysis-based transformations was recognized and addressed in the 1980s [27]. The correctness proofs that we know of, notably those of Wand and his coauthors [39, 42, 43], are based on special-purpose operational semantics, and prove only one direction of correctness: if the source program evaluates to a final answer, then so does the transformed program (this is often called soundness). In this paper we give alternative and more general methods for proving the correctness of CFA-based transformations, based on a simple denotational semantics and principles from data abstraction.

We begin by describing a simply typed, call-by-value functional language that has control-flow information incorporated into types (Section 2). The type system is essentially the same as Heintze's [15]. We then show how to translate that language, in a type-directed way, into a first-order programming language (Sections 3 and 4). We call this translation *CFA-based defunctionalization* [7, 41]; it refines Reynolds's defunctionalization method for turning higher-order programs into first-order programs [32, 34, 35]. The first-order target language is Pascal-like and includes rudimentary support for abstract data types as well as support for nested function definitions. The main results of the paper are the following:

- We prove CFA-based defunctionalization correct using a denotational semantics for the target language (Section 4). That is, we prove that a source program evaluates to a final answer iff the defunctionalized program evaluates to the same final answer.
- Using the semantics of the target, we prove the correctness of flow-based inlining (Section 5), a source-to-source transformation.
- We augment the CFA-based type system and give a new, type-based specification of "lightweight defunctionalization" (Section 6), the analog of "lightweight closure conversion" [39]. We also show how to transform new type system into the first-order target language, and prove correctness using principles from data abstraction.

The second and third results both use the first result, albeit in different ways. We expect that other correctness results can reuse the correctness theorem for CFA-based defunctionalization.

To illustrate how correctness proofs go, consider the following example of flow-based inlining. This example is adapted from Shivers's thesis [38], and is written in ML-style syntax.

```
let fun g x = x in
let fun f h y =
    (if (y=0) then (h 1)
    else f h (h y)) in
f g 2
```

Source program

```
let fun g x = x in
let fun f h y =
    (if (y=0) then (g 1)
     else f g (g y)) in
f g 2
```

Transformed program

CFA predicts that the only function that can flow to `h` is `g`, and thus `h` can be replaced by `g`. (A separate transformation, useless variable elimination [22, 38, 44], could be used to remove `h` from `f`.) One might expect that β-reduction or β-equality could be used to prove that the source and transformed programs are the same. However, the source and transformed programs are *not* equivalent via β-reduction or β-equality, because of the recursive definition of `f`. In Section 5, we shall prove the two programs equivalent by showing that their defunctionalized versions are equivalent. The correctness then follows from the correctness of defunctionalization (Section 4).

2 The Source Language

The source language, PCFA, is a version of PCF [31, 36] with control-flow types [15]. Each function definition in a program has a unique label; we use ℓ to range over these labels. Each call site has a set of labels representing the functions that may flow to that site; we use L to range over finite sets of labels. Function types include these label sets, and the type system ensures that the functions that reach call sites have labels in the label set associated with the call site.

More specifically, the types and terms of the language are given by following grammar, where $\texttt{op}$ abbreviates either the $\texttt{succ}$ or the $\texttt{pred}$ operations.

$$
\begin{aligned}
s &::= \texttt{int} \mid (s \to s, L) \\
D &::= (\texttt{fun}^{\ell}\, f\, (x : s) = e) \\
e &::= n \mid x \mid (e_1 \;@_L\; e_2) \mid (\texttt{let}\; D\; \texttt{in}\; e) \mid (\texttt{let}\;\texttt{val}\; x = e\; \texttt{in}\; e) \\
&\quad \mid (\texttt{op}\; e) \mid (\texttt{if0}\; e\; \texttt{then}\; e_1\; \texttt{else}\; e_2)
\end{aligned}
$$

The symbols x, f range over ordinary variables, n over integers, s, t over types, and e over terms. A recursive function f with argument x is defined using the construct $(\texttt{fun}^{\ell}\, f\, (x : s) = e)$, where e is the body of the recursive definition. Following standard conventions [5], we identify terms up to renaming of bound variables and use the notation $e[x := e']$ for the capture-free substitution of e' for x in e. Finally, we often omit types when they can be reconstructed.

The type rules for PCFA appear in Table 1, and include five forms of judgement. The rules use two different contexts, one denoted ψ and the other denoted Γ. The context ψ is a label context: it maps a function label ℓ to a triple (Γ, s, t), where Γ is the type context in which the function labelled ℓ is defined, and s and t are respectively the argument and return types of the function. The second form of context, Γ, is a list of assignments of types to variables. We will use the notation $x_1 : s_1, \ldots, x_n : s_n$ to denote such type contexts. Standard presentations of type-based CFA [15] use only the type context Γ. As the analysis is a whole-program analysis, the label context ψ can be used to provide information about which functions can be legally used at a program point.

For example, consider the rule for recursive function declaration. The label ℓ occurs in the label context ψ. The type context, Γ, used to define the recursive function makes sense because it is exactly the definition type context of the recursive function stipulated in the side condition $\psi(\ell) = (\Gamma, s, t)$.

The operational semantics of PCFA appears in Table 2 using a natural semantics [21]. The form of the judgement is $(e, \gamma) \Downarrow v$, where γ is a finite function from variables to values, v ranges over values, and the set of values includes the numerals n and closures of the form $\langle \texttt{fun}^{\ell}\, f\, (x : s) = e, \gamma \rangle$. In the rules, $\dot{-}$ denotes the operation that subtracts one unless its argument is 0, in which case it returns 0. Most of the rules are self-explanatory. One can prove the following theorem:

Theorem 1 (Subject Reduction). *Suppose $\Psi \mid \emptyset \vdash e : s$ and $(e, \emptyset) \Downarrow v$. Then $\Psi \mid \emptyset \vdash v : s$ is derivable.*

This theorem encompasses the usual correctness criterion for control-flow analysis: a function labelled ℓ called at an application site labelled L is such that $\ell \in L$.

Table 1. Typing rules for the source language.

Well-formed types $\boxed{\psi \vdash s}$

$$\psi \vdash \texttt{int} \qquad \frac{\psi \vdash s \qquad \psi \vdash t \qquad L \subseteq dom(\psi)}{\psi \vdash (s \to t, L)}$$

Subtyping $\boxed{\psi \vdash s \leq t}$

$$\frac{\psi \vdash s}{\psi \vdash s \leq s} \qquad \frac{\psi \vdash s_1' \leq s_1 \qquad \psi \vdash s_2 \leq s_2' \qquad L \subseteq L' \subseteq dom(\psi)}{\psi \vdash (s_1 \to s_2, L) \leq (s_1' \to s_2', L')}$$

Well-formed contexts $\boxed{\psi \mid \Gamma \vdash \diamond}$

$$\frac{\forall \ell \text{ such that } \psi(\ell) = ((x_1 : s_1, \ldots, x_n : s_n), s_{n+1}, s_{n+2}).\psi \vdash s_i}{\psi \mid \emptyset \vdash \diamond} \qquad \frac{\psi \mid \Gamma \vdash \diamond \quad x \notin \Gamma \quad \psi \vdash s}{\psi \mid \Gamma, x : s \vdash \diamond}$$

Well-formed declarations $\boxed{\psi \mid \Gamma \vdash D \triangleright \Gamma'}$

$$\frac{\psi \mid \Gamma_i, f : (s_i \to t_i, \{\ell_i\}), x : s_i \vdash e : t_i \qquad f, x \notin \Gamma_i}{\psi \mid \Gamma_i \vdash (\texttt{fun}^{\ell_i}\, f\ (x : s_i) = e) \triangleright (f : (s_i \to t_i, \{\ell_i\}))} \quad \psi(\ell_i) = (\Gamma_i, s_i, t_i)$$

Well-formed expressions $\boxed{\psi \mid \Gamma \vdash e : s}$

$$\frac{\psi \mid \Gamma \vdash e : s \qquad \psi \vdash s \leq s'}{\psi \mid \Gamma \vdash e : s'} \qquad \frac{\psi \mid \Gamma \vdash e : \texttt{int} \qquad \psi \mid \Gamma \vdash e_i : s}{\psi \mid \Gamma \vdash (\texttt{if0}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) : s}$$

$$\frac{\psi \mid \Gamma, x : s, \Gamma' \vdash \diamond}{\psi \mid \Gamma, x : s, \Gamma' \vdash x : s} \qquad \frac{\psi \mid \Gamma \vdash D \triangleright \Gamma' \qquad \psi \mid \Gamma, \Gamma' \vdash e : s}{\psi \mid \Gamma \vdash \texttt{let}\ D\ \texttt{in}\ e : s}$$

$$\frac{\psi \mid \Gamma \vdash \diamond}{\psi \mid \Gamma \vdash n : \texttt{int}} \qquad \frac{\psi \mid \Gamma \vdash e' : t \qquad \psi \mid \Gamma, x : t \vdash e : s \qquad x \notin \Gamma}{\psi \mid \Gamma \vdash \texttt{let}\ \texttt{val}\ x = e'\ \texttt{in}\ e : s}$$

$$\frac{\psi \mid \Gamma \vdash e : \texttt{int}}{\psi \mid \Gamma \vdash (\texttt{op}\ e) : \texttt{int}} \qquad \frac{\psi \mid \Gamma \vdash e_1 : (s \to t, L) \quad \psi \mid \Gamma \vdash e_2 : s}{\psi \mid \Gamma \vdash e_1\ @_L\ e_2 : t} \quad L \subseteq dom(\psi)$$

Table 2. Operational semantics for the source language.

$$(n,\gamma) \Downarrow n \qquad\qquad (x,\gamma) \Downarrow \gamma(x)$$

$$\frac{(e,\gamma) \Downarrow n}{(\texttt{succ}\, e,\gamma) \Downarrow (n+1)} \qquad\qquad \frac{(e,\gamma) \Downarrow n}{(\texttt{pred}\, e,\gamma) \Downarrow (n \dot{-} 1)}$$

$$\frac{(e,\gamma) \Downarrow 0 \qquad (e_1,\gamma) \Downarrow v}{(\texttt{if0}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2,\gamma) \Downarrow v} \qquad\qquad \frac{(e,\gamma) \Downarrow (n+1) \qquad (e_2,\gamma) \Downarrow v}{(\texttt{if0}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2,\gamma) \Downarrow v}$$

$$\frac{(e',\gamma[f \mapsto \langle \texttt{fun}^\ell\, f\,(x:s) = e,\gamma\rangle]) \Downarrow v}{(\texttt{let fun}^\ell\, f\,(x:s) = e\, \texttt{in}\, e',\gamma) \Downarrow v} \qquad\qquad \frac{(e,\gamma) \Downarrow v' \qquad (e',\gamma[x \mapsto v']) \Downarrow v}{(\texttt{let val}\, x = e\, \texttt{in}\, e',\gamma) \Downarrow v}$$

$$\frac{(e,\gamma) \Downarrow \langle \texttt{fun}^\ell\, f\,(x:s) = e'',\gamma' \rangle = v_0 \qquad (e',\gamma) \Downarrow v' \qquad (e'',\gamma'[f \mapsto v_0, x \mapsto v']) \Downarrow v}{((e\, @_L\, e'),\gamma) \Downarrow v}\ \ell \in L$$

3 The Target Language

The target language of defunctionalization is a first-order language with rudimentary datatype support. The syntax of the target language is given by the following grammar.

$$\begin{aligned}
s &::= \texttt{unit} \mid \texttt{int} \mid (s \times s) \mid (\alpha_1 \cup \ldots \cup \alpha_n) \\
D &::= \texttt{tagtype}\, \alpha = C\, \texttt{of}\, s \mid \texttt{fun}\, f\,(x:s) = e \\
e &::= x \mid n \mid (\texttt{op}\, e) \mid (\texttt{if0}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2) \mid \langle\rangle \mid \langle e_1, e_2 \rangle \\
&\quad \mid (\texttt{proj}_1\, e) \mid (\texttt{proj}_2\, e) \mid f(e) \mid C(e) \\
&\quad \mid (\texttt{case}\, e\, \texttt{of}\, C_1(x) \Rightarrow e_1 \mid \ldots \mid C_n(x) \Rightarrow e_n) \\
&\quad \mid (\texttt{let}\, D \ldots D\, \texttt{in}\, e) \mid (\texttt{let val}\, x = e\, \texttt{in}\, e)
\end{aligned}$$

The symbols α, β range over type variables and C over constructors. Types in the target language include product types and union types but not function types. As in Pascal, there is a means of declaring functions inside other functions; we will use this feature in our translation of lightweight defunctionalization in Section 6.

To make programs in the target simpler to read, we adopt a few notational conventions. Terms are identified up to renaming of bound variables; note that the expression $(\texttt{tagtype}\, \alpha = C\, \texttt{of}\, s)$ binds both α and C. Product types are extended to $(s_1 \times \ldots \times s_n)$, with the convention that such types associate to the right; we also use the notation $(\texttt{proj}_i\, e)$ for projections from these expanded types, with tuples extended correspondingly, and use pattern-matching syntax in function declarations instead of projections. Union types that are the same except for the order are regarded as equal types.

The type rules for the target language appear in Tables 3 and 4. In these rules, type contexts Δ are lists of type variables and bindings of constructors and ordinary variables to "extended types" σ, which are ordinary types s or function types $(s \to t)$. Note that the extended function types are "second-class": functions with such types can not accept functions as arguments and return functions as results. Also, note that in the rule for

Table 3. Typing rules for the target language.

Well-formed types $\boxed{\Delta \vdash s}$

$$\frac{\Delta \vdash \diamond}{\Delta \vdash \mathtt{int}} \qquad \frac{\Delta \vdash \diamond}{\Delta \vdash \mathtt{unit}} \qquad \frac{\Delta \vdash \diamond \qquad \alpha_i \in \Delta}{\Delta \vdash \alpha_1 \cup \ldots \cup \alpha_n} \qquad \frac{\Delta \vdash s \qquad \Delta \vdash t}{\Delta \vdash (s \times t)}$$

Subtyping $\boxed{\Delta \vdash s \leq t}$

$$\frac{\Delta \vdash s}{\Delta \vdash s \leq s} \qquad \frac{\Delta \vdash (\alpha_1 \cup \ldots \cup \alpha_n) \quad \{j_1, \ldots, j_k\} \subseteq \{1, \ldots, n\}}{\Delta \vdash (\alpha_{j_1} \cup \ldots \cup \alpha_{j_k}) \leq (\alpha_1 \cup \ldots \cup \alpha_n)}$$

$$\frac{\Delta \vdash s \leq s' \qquad \Delta \vdash s' \leq s''}{\Delta \vdash s \leq s''} \qquad \frac{\Delta \vdash s_i \leq s'_i}{\Delta \vdash (s_1 \times \ldots \times s_n) \leq (s'_1 \times \ldots \times s'_n)}$$

Well-formed contexts $\boxed{\Delta \vdash \diamond}$

$$\emptyset \vdash \diamond \qquad \frac{\Delta \vdash \diamond \qquad \alpha \notin \Delta}{\Delta, \alpha \vdash \diamond} \qquad \frac{\Delta \vdash \diamond \qquad \Delta \vdash s \qquad x \notin \Delta}{\Delta, x : s \vdash \diamond}$$

$$\frac{\Delta \vdash \alpha \qquad \Delta \vdash s \qquad C \notin \Delta}{\Delta, C : s \to \alpha \vdash \diamond} \qquad \frac{\Delta \vdash \diamond \qquad \Delta \vdash s \qquad \Delta \vdash t \qquad f \notin \Delta}{\Delta, f : s \to t \vdash \diamond}$$

Well-formed declarations $\boxed{\Delta \vdash D \triangleright \Delta'}$

$$\frac{\Delta, \alpha, C : s \to \alpha, \Delta' \vdash \diamond}{\Delta, \alpha, C : s \to \alpha, \Delta' \vdash (\mathtt{tagtype}\ \alpha\ =\ C\ \mathtt{of}\ s) \triangleright (\alpha, C : (s \to \alpha))}$$

$$\frac{\Delta, f : (s \to t), \Delta', x : s \vdash e : t \qquad f, x \notin \Delta, \Delta'}{\Delta, f : (s \to t), \Delta' \vdash (\mathtt{fun}\ f\ (x : s) = e) \triangleright (f : s \to t)}$$

Table 4. Typing rules for the target language, continued.

Well-formed expressions

$\boxed{\Delta \vdash e : s}$

$$\frac{\Delta, x : s, \Delta' \vdash \diamond}{\Delta, x : s, \Delta' \vdash x : s} \qquad \frac{\Delta \vdash \diamond}{\Delta \vdash n : \texttt{int}}$$

$$\frac{\Delta \vdash e : \texttt{int}}{\Delta \vdash (\texttt{op}\, e) : \texttt{int}} \qquad \frac{\Delta \vdash e : \texttt{int} \qquad \Delta \vdash e_i : s}{\Delta \vdash (\texttt{if0}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2) : s}$$

$$\frac{\Delta \vdash \diamond}{\Delta \vdash \langle\rangle : \texttt{unit}} \qquad \frac{\Delta, f : (s \to t), \Delta' \vdash e : s}{\Delta, f : (s \to t), \Delta' \vdash f(e) : t}$$

$$\frac{\Delta \vdash e_i : s_i}{\Delta \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)} \qquad \frac{\Delta \vdash e : (s_1 \times s_2)}{\Delta \vdash (\texttt{proj}_i\, e) : s_i}$$

$$\frac{\Delta, \alpha, C : (s \to \alpha), \Delta' \vdash e : s}{\Delta, \alpha, C : (s \to \alpha), \Delta' \vdash C(e) : \alpha} \qquad \frac{\Delta \vdash e : s \qquad \Delta \vdash s \leq s'}{\Delta \vdash e : s'}$$

$$\frac{\Delta \vdash e : (\alpha_1 \cup \ldots \cup \alpha_n) \qquad \Delta, x : s_i \vdash e_i : s \qquad \Delta(C_i) = (s_i \to \alpha_i)}{\Delta \vdash (\texttt{case}\, e\, \texttt{of}\, C_1(x) \Rightarrow e_1 \mid \ldots \mid C_n(x) \Rightarrow e_n) : s}$$

$$\frac{\Delta, \Delta_1, \ldots, \Delta_n \vdash D_i \rhd \Delta_i, \text{ for all } i \qquad \Delta, \Delta_1, \ldots, \Delta_n \vdash e : s}{\Delta \vdash (\texttt{let}\, D_1 \ldots D_n\, \texttt{in}\, e) : s}$$

$$\frac{\Delta \vdash e' : t \qquad \Delta, x : t \vdash e : s \qquad x \notin \Delta}{\Delta \vdash (\texttt{let}\, \texttt{val}\, x = e'\, \texttt{in}\, e) : s}$$

Table 5. Operational semantics for the target language.

$$(n,\delta) \Downarrow n \qquad\qquad (\langle\rangle,\delta) \Downarrow \langle\rangle$$

$$\frac{(e,\delta) \Downarrow n}{(\mathtt{succ}\, e,\delta) \Downarrow (n+1)} \qquad\qquad \frac{(e,\delta) \Downarrow n}{(\mathtt{pred}\, e,\delta) \Downarrow (n\dot{-}1)}$$

$$\frac{(e,\delta) \Downarrow 0 \quad (e_1,\delta) \Downarrow v}{(\mathtt{if0}\; e\; \mathtt{then}\; e_1\; \mathtt{else}\; e_2,\delta) \Downarrow v} \qquad\qquad \frac{(e,\delta) \Downarrow (n+1) \quad (e_2,\delta) \Downarrow v}{(\mathtt{if0}\; e\; \mathtt{then}\; e_1\; \mathtt{else}\; e_2,\delta) \Downarrow v}$$

$$\frac{(e_i,\delta) \Downarrow v_i}{(\langle e_1,e_2\rangle,\delta) \Downarrow \langle v_1,v_2\rangle} \qquad\qquad \frac{(e,\delta) \Downarrow \langle v_1,v_2\rangle}{(\mathtt{proj}_i\; e,\delta) \Downarrow v_i}$$

$$\frac{(e,\delta) \Downarrow C_i(v') \quad (e_i,\delta[x \mapsto v']) \Downarrow v}{(\mathtt{case}\; e\; \mathtt{of}\; C_1(x) \Rightarrow e_1 \mid \ldots \mid C_n(x) \Rightarrow e_n,\delta) \Downarrow v} \qquad\qquad \frac{(e,\delta) \Downarrow v}{(C(e),\delta) \Downarrow C(v)}$$

$$(x,\delta) \Downarrow \delta(x) \qquad\qquad \frac{(e,\delta) \Downarrow v}{(\mathtt{let}\;\; \mathtt{in}\; e,\delta) \Downarrow v}$$

$$\frac{(\mathtt{let}\; D_1 \ldots D_n\; \mathtt{in}\; e,\delta) \Downarrow v}{(\mathtt{let}\; \mathtt{tagtype}\; \alpha_1 \;=\; C_1\; \mathtt{of}\; s_1\; D_1 \ldots D_n\; \mathtt{in}\; e,\delta) \Downarrow v}$$

$$\frac{(\mathtt{let}\; D_1 \ldots D_n\; \mathtt{in}\; e,\delta[f \mapsto \mathtt{fun}\; f\; (x:s) = e']) \Downarrow v}{(\mathtt{let}\; \mathtt{fun}\; f\; (x:s) = e'\; D_1 \ldots D_n\; \mathtt{in}\; e,\delta) \Downarrow v}$$

$$\frac{\delta(f) = (\mathtt{fun}\; f\; (x:s) = e') \quad (e,\delta) \Downarrow v' \quad (e',\delta[x \mapsto v']) \Downarrow v}{(f(e),\delta) \Downarrow v}$$

$$\frac{(e',\delta) \Downarrow v' \quad (e,\delta[x \mapsto v']) \Downarrow v}{(\mathtt{let}\; \mathtt{val}\; x = e'\; \mathtt{in}\; e,\delta) \Downarrow v}$$

`let`, the declarations $D_1,\ldots,D_n$ are possibly mutually recursive. This explains the use of type contexts $\Delta_1,\ldots,\Delta_n$ in the antecedent of the rule for `let`.

The operational semantics of the target language appears in Table 5. We use the same style as the operational semantics for PCFA, i.e., a natural semantics, and write $(e,\delta) \Downarrow v$ when the term e, in environment δ, returns the final value v of the form n, $\langle\rangle$, $\langle v_1,v_2\rangle$, or $C(v)$. It is straightforward to prove the following theorem:

Theorem 2 (Subject Reduction). *Suppose $\emptyset \vdash e : s$ and $(e,\emptyset) \Downarrow v$. Then $\emptyset \vdash v : s$ is derivable.*

In addition to the operational semantics, we also define a denotational semantics of the target. The denotational semantics—with its emphasis on extensional functions and interpretation of recursion as least-fixed-point—simplifies some of the proofs, especially those involving data abstraction. The denotational semantics of the target language is set-theoretic: no domain theory or posets are needed due to the lack of higher-order functions. It is given in two parts: a meaning function for types and a meaning function for terms. Since types involve type variables, the meaning function for types requires a notion of type environment. Suppose $\Delta \vdash \diamond$ and η is a finite partial function from type variables to sets. We say η is **type environment for** Δ if

- For any $\alpha \in \Delta$, $\alpha \in dom(\eta)$, and
- If $\alpha_0 \neq \alpha_1$ with $\eta(\alpha_i)$ defined, then $\eta(\alpha_0)$ and $\eta(\alpha_1)$ are disjoint sets.

Note that

Proposition 1. *For any $\Delta \vdash \diamond$, there is a canonical type environment η for Δ.*

The proof uses standard set-theoretic techniques. For example, if

$$\Delta = \alpha, C : (\mathtt{int} \times \alpha) \cup \mathtt{unit} \to \alpha,$$

the canonical η assigns α to finite lists of integers. For a type environment η, and a type s, we define $[\![\Delta \vdash s]\!]\eta$ as follows:

$$\begin{array}{ll}
[\![\Delta \vdash \mathtt{unit}]\!]\eta & = \{\langle\rangle\} \\
[\![\Delta \vdash \mathtt{int}]\!]\eta & = \mathbf{N} \\
[\![\Delta \vdash \alpha_1 \cup \ldots \cup \alpha_n]\!]\eta & = \eta(\alpha_1) \cup \ldots \cup \eta(\alpha_n) \\
[\![\Delta \vdash s \times s']\!]\eta & = [\![\Delta \vdash s]\!]\eta \times [\![\Delta \vdash s']\!]\eta
\end{array}$$

Since terms involve variables too, the meaning of terms also requires a notion of environment. An (ordinary) **environment** ρ is a finite partial function from variables to elements of sets or to partial functions. Just like type environments, there are some well-formedness conditions on environments. Suppose ρ is an environment, and η is a type environment compatible with Δ. We say that ρ is **compatible with** η, Δ if for any $x \in dom(\Delta)$,

- If $x : s \in \Delta$, then $\rho(x) \in [\![\Delta \vdash s]\!]\eta$;
- If $(f : s \to t) \in \Delta$, then $\rho(f)$ is a partial function from $[\![\Delta \vdash s]\!]\eta$ to $[\![\Delta \vdash t]\!]\eta$; and
- If $(C : s \to \alpha) \in \Delta$, then $\rho(C)$ is an isomorphism from $[\![\Delta \vdash s]\!]\eta$ to $\eta(\alpha)$.

Suppose ρ is compatible with η, Δ. The meaning of a judgement $\Delta \vdash e : s$ in the target language can be defined by induction on the typing derivation. The semantics of terms is standard (see *e.g.*, Gunter's text [13] or Mitchell's text [24]). For space reasons, we omit most of the clauses and concentrate on the meaning of `let`. Suppose the term is $e = (\mathtt{let}\ D_1 \ldots D_n\ \mathtt{in}\ e')$, and we want to interpret it in the type environment η and ordinary environment ρ. Let $E_1, \ldots, E_m$ be the type declarations (each of the form $(\mathtt{tagtype}\ \alpha_i\ =\ C_i\ \mathtt{of}\ s_i)$) and let $F_1, \ldots, F_k$ be the function declarations (each of the form $(\mathtt{fun}\ f_i\ (x : s_i) = e_i)$). Let η' be the canonical extension of η compatible with

$$\Delta' = \Delta, \alpha_1, C_1 : s_1 \to \alpha_1, \ldots, \alpha_m, C_m : s_m \to \alpha_m, f_1 : s_1 \to t_1, \ldots, f_k : s_k \to t_k$$

(which must exist by the Proposition above). Let h_i be the isomorphism from $[\![\Delta \vdash s_i]\!]\eta$ to $\eta(\alpha_i)$, and let $g_1, \ldots, g_n$ be the least partial functions such that for all i and any $d \in [\![\Delta' \vdash s_i]\!]\eta'$,

$$g_i(d) = [\![\Delta', x : s_i \vdash e_i : s_i']\!]\eta'\ \rho'[x \mapsto d]$$

where

$$\rho' = \rho[C_1 \mapsto h_1, \ldots, C_m \mapsto h_m, f_1 \mapsto g_1, \ldots, f_n \mapsto g_n]$$

Then define

$$[\![\Delta \vdash e : s]\!]\eta\rho \simeq [\![\Delta' \vdash e' : s]\!]\eta'\,\rho'$$

where $\simeq$ means that the left side is defined and equal to the right side iff the right side is defined.

The following theorem shows that we may use the denotational semantics to reason safely about programs:

Theorem 3 (Adequacy). *Suppose* $\emptyset \vdash e : \mathtt{int}$. *Then* $[\![\emptyset \vdash e : \mathtt{int}]\!]\emptyset\emptyset = n$ *iff* $(e,\emptyset) \Downarrow n$.

Proof. (Sketch) The $(\Leftarrow)$ direction relies on an induction on the proof of $(e,\emptyset) \Downarrow n$. The $(\Rightarrow)$ direction, on the other hand, relies on a lexicographic induction on (k,e), where k denotes a finite number of unwindings of recursive function definitions.

4 Defunctionalization

We next show how to translate the source language to the target, and prove that the translation is correct.

The translation is a modified version of Reynolds's defunctionalization [32, 34] that takes advantage of control-flow information. In defunctionalization, functions in the source are represented in the target as tagged tuples, where the tag indicates the code of the closure, and the tuple represents the values of the free variables of the closure. Function application in the source must therefore be made explicit in the target via an *apply* function. Specifically, if f_t is the representation in the target of a source function f_s, and a_t is the representation in the target of an argument a_s of f_s, then the result of $apply(f_t, a_t)$ will be the representation in the target of the result of $f_s(a_s)$ [32, 34].

Before giving the details of the translation, we give an example to show how all of the elements of the target language are used in the translation. Consider the following implementation of sets-as-characteristic functions in ML:

```
let fun empty x = false in
let fun insert x s =
    let fun test(y) = (x=y) || (s y) in test in
let fun member x s = s x in
...
```

When rendered in a simple extension of PCFA with booleans, the above code becomes

```
let funℓ1 empty (x:int) = false in
let funℓ2 insert(x:int) =
   (let funℓ3 check(s:u) =
    let funℓ4 test(y:int) = (x = y) || (s @{ℓ1,ℓ4} y) in test
    in check) in
let funℓ5 member(x:int) =
   (let funℓ6 check'(s:u) = s @{ℓ1,ℓ4} x in check') in
...
```

where u is the type (int→bool, $\{\ell_1,\ell_4\}$).

The representation of functions in the target language is based on abstract types and constructor functions. We translate each function label into an abstract type declaration, where the concrete representation type reflects the types of the free variables. For the functions empty, insert and member, we set

```
tagtype α1 = C1 of unit
tagtype α2 = C2 of α1
tagtype α5 = C5 of α1 × α2
```

since insert lies in the scope of empty and member lies in the scope of empty and insert. (A less simplistic defunctionalization would take into account only free variables.) Likewise, the corresponding types for the functions check, test, and check′ are

```
tagtype α3 = C3 of α1 × α2×int
tagtype α4 = C4 of α1 × α2×int×α3 × (α1 ∪ α4)
tagtype α6 = C6 of α1 × α2 × α5×int
```

Note that the definition of α_4 is recursive: since s is a free variable of test, and since the type of s in the source is (int → int, $\{\ell_1,\ell_4\}$), the type translation of s in the target is the disjoint union type $\alpha_1 \cup \alpha_4$, making α_4's definition recursive.

Function application in the source is defined via *apply* functions in the target. A standard defunctionalization would yield the following monolithic version of *apply*.

```
fun apply⟨F,a⟩ =
  case F of
     (empty as C1⟨⟩) => false
   | (insert as C2⟨d⟩) =>
     let val check = C3⟨d,insert,a⟩ in check
   | (check as C3⟨d,e,x⟩) =>
     let val test  = C4⟨d,e,x,check,a⟩ in test
   | (test as C4⟨d,e,x,v,s⟩) =>
     (x = a) || apply(s,a)
   | (member as C5⟨d,e⟩) =>
     let val check′= C6⟨d,e,member,a⟩ in check′
   | (check′ as C6⟨d,e,v,x⟩) => apply(a,x)
```

Using the control-flow types, we can split the monolithic *apply* function into multiple (possibly mutually recursive) application functions, one for each function label [7,41]. Thus, the above monolithic apply function can be split as

```
fun apply1(empty as C1⟨⟩, a)  = false
fun apply2(insert as C2⟨d⟩,a) =
  let val check = C3⟨d,insert,a⟩ in check
fun apply3(check as C3⟨d,e,x⟩,a)=
  let val test = C4⟨d,e,x,check,a⟩ in test
fun apply4(test as C4⟨d,e,x,v,s⟩,a)=
  (x = a) || (case s of
                 (empty as C1⟨⟩) => apply1(empty,a)
               | (test  as C4⟨d,e,x,v,s⟩) => apply4(test,a))
  ...
```

Note how the case analysis occurs only at the application points. Also, note that while the *source program has no recursive functions, the target program has a recursive function*, namely, `apply4`. This is one source of complication in proving the correctness of defunctionalization.

To treat the representation of function types more abstractly, it is helpful to use functions to construct values:

```
fun make1()       = C1⟨⟩          fun make4(d,e,x,v,a)= C4⟨d,e,x,v,a⟩
fun make2(d)      = C2⟨d⟩         fun make5(d,e)      = C5⟨d,e⟩
fun make3(d,e,x)= C3⟨d,e,x⟩     fun make6(d,e,v,x)  = C6⟨d,e,v,x⟩
```

This is a key idea: the `make` functions for different program transformations, *e.g.*, lightweight defunctionalization (Section 6), will use different concrete representations of functions in the source program. The complete translation of the example appears in Figure 1.

```
let tagtype α1 = C1 of unit
    tagtype α2 = C2 of α1
    tagtype α3 = C3 of α1 × α2 × int
    tagtype α4 = C4 of α1 × α2 × int × α3 × (α1 ∪ α4)
    tagtype α5 = C5 of α1 × α2
    tagtype α6 = C6 of α1 × α2 × α5 × int
    fun make1()          = C1⟨⟩
    fun make2(d)         = C2⟨d⟩
    fun make3(d,e,x)     = C3⟨d,e,x⟩
    fun make4(d,e,x,v,a) = C4⟨d,e,x,v,a⟩
    fun make5(d,e)       = C5⟨d,e⟩
    fun make6(d,e,v,x)   = C6⟨d,e,v,x⟩
    fun apply1(empty as C1⟨⟩, a)  = false
    fun apply2(insert as C2⟨d⟩,a) =
      let val check = make3(d,insert,a) in check
    fun apply3(check as C3⟨d,e,x⟩,a)=
      let val test = make4(d,e,x,check,a) in test
    fun apply4(test as C4⟨d,e,x,v,s⟩,a)=
      (x = a) || (case s of
                   (empty as C1⟨⟩) => apply1(empty,a)
                 | (test  as C4⟨d,e,x,v,s⟩) => apply4(test,a))
      ...
in let val empty  = make1() in
   let val insert = make2(empty) in
   let val member = make5(empty,insert) in
...
```

Fig. 1. Defunctionalization of sets-as-functions example.

Table 6. Defunctionalization as a type-directed translation.

Declarations $\boxed{(\psi \mid \Gamma \vdash D \rhd \Gamma') \Rightarrow (\psi^*, \Gamma^* \vdash D^* : \alpha; D')}$

$$\frac{\psi \mid \Gamma, f : (s \to t, \{\ell\}), x : s \vdash e : t \qquad f, x \notin \Gamma}{\psi \mid \Gamma \vdash (\mathtt{fun}^{\ell}\, f\,(x : s) = e) \rhd (f : s \to t, \{\ell\})} \Rightarrow \frac{\psi^*, \Gamma^*, f : \alpha, x : s^* \vdash e^* : t^* \;;\; D}{\psi^*, \Gamma^* \vdash \mathit{make}\langle x_1, \ldots, x_n\rangle : \alpha \;;\; D \cup D'}$$

$$\text{where } \begin{cases} \psi(\ell) = (\Gamma, s, t) \\ \Gamma = \langle x_1 : u_1, \ldots, x_n : u_n\rangle \\ D' = \{\ \mathtt{tagtype}\,\alpha\ =\ C\ \mathtt{of}\,\Gamma^*, \\ \qquad \mathtt{fun}\,\mathit{make}\ (\langle x_1, \ldots, x_n\rangle) = C\langle x_1, \ldots, x_n\rangle, \\ \qquad \mathtt{fun}\,\mathit{apply}\ (f\ \mathtt{as}\ C\langle x_1, \ldots, x_n\rangle, x) = e^*\} \end{cases}$$

Expressions $\boxed{(\psi \mid \Gamma \vdash e : s) \Rightarrow (\psi^*, \Gamma^* \vdash e^* : s^*; D)}$

$$\frac{\psi \mid \Gamma, x : s, \Gamma' \vdash \diamond}{\psi \mid \Gamma, x : s, \Gamma' \vdash x : s} \Rightarrow \frac{\psi^*, \Gamma^*, x : s^*, (\Gamma')^* \vdash \diamond}{\psi^*, \Gamma^*, x : s^*, (\Gamma')^* \vdash x : s^* \;;\; \emptyset}$$

$$\frac{\psi \mid \Gamma \vdash \diamond}{\psi \mid \Gamma \vdash n : \mathtt{int}} \Rightarrow \frac{\psi^*, \Gamma^* \vdash \diamond}{\psi^*, \Gamma^* \vdash n : \mathtt{int} \;;\; \emptyset}$$

$$\frac{\psi \mid \Gamma \vdash e : \mathtt{int}}{\psi \mid \Gamma \vdash (\mathtt{op}\, e) : \mathtt{int}} \Rightarrow \frac{\psi^*, \Gamma^* \vdash e^* : \mathtt{int} \;;\; D}{\psi^*, \Gamma^* \vdash (\mathtt{op}\, e^*) : \mathtt{int} \;;\; D}$$

$$\frac{\psi \mid \Gamma \vdash e : \mathtt{int} \qquad \psi \mid \Gamma \vdash e_i : s}{\psi \mid \Gamma \vdash (\mathtt{if0}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2) : s} \Rightarrow \frac{\psi^*, \Gamma^* \vdash e^* : \mathtt{int} \;;\; D \qquad \psi^*, \Gamma^* \vdash e_i^* : s^* \;;\; D_i}{\psi^*, \Gamma^* \vdash (\mathtt{if0}\, e^*\, \mathtt{then}\, e_1^*\, \mathtt{else}\, e_2^*) : s^* \;;\; D \cup D_i}$$

$$\frac{\psi \mid \Gamma \vdash e' : t \;\; \psi \mid \Gamma, x : t \vdash e : s \;\; x \notin \Gamma}{\psi \mid \Gamma \vdash \mathtt{let\ val}\, x = e'\, \mathtt{in}\, e : s} \Rightarrow \frac{\psi^*, \Gamma^* \vdash (e')^* : t^* \;;\; D_1 \qquad \psi^*, \Gamma^*, x : t^* \vdash e^* : s^* \;;\; D_2}{\psi^*, \Gamma^* \vdash \mathtt{let\ val}\, x = (e')^*\, \mathtt{in}\, e^* : s^* \;;\; D_1 \cup D_2}$$

$$\frac{\psi \mid \Gamma \vdash e : s \quad \psi \mid \Gamma \vdash s \leq t}{\psi \mid \Gamma \vdash e : t} \Rightarrow \frac{\psi^*, \Gamma^* \vdash e^* : s^*; D \quad \psi^*, \Gamma^* \vdash s^* \leq t^*}{\psi^*, \Gamma^* \vdash e^* : t^*; D}$$

$$\frac{\psi \mid \Gamma \vdash D \rhd \Gamma' \quad \psi \mid \Gamma, \Gamma' \vdash e : s}{\psi \mid \Gamma \vdash \mathtt{let}\, D\, \mathtt{in}\, e : s} \Rightarrow \frac{\psi^*, \Gamma^* \vdash D^* : \alpha_i \;;\; D_1 \quad \psi^*, \Gamma^*, (\Gamma')^* \vdash e^* : s^* \;;\; D_2}{\psi^*, \Gamma^* \vdash \mathtt{let\ val}\, f = D^*\, \mathtt{in}\, e^* : s^* \;;\; D_1 \cup D_2}$$

where $D = (\mathtt{fun}^{\ell_i}\, f\,(x : s_i) = e')$

$$\frac{\psi \mid \Gamma \vdash e_1 : (s \to t, L) \quad \psi \mid \Gamma \vdash e_2 : s}{\psi \mid \Gamma \vdash e_1 \;@_L\; e_2 : t} \Rightarrow \frac{\psi^*, \Gamma^* \vdash e_1^* : \alpha_{i_1} \cup \ldots \;;\; D_1 \quad \psi^*, \Gamma^* \vdash e_2^* : s^* \;;\; D_2}{\psi^*, \Gamma^* \vdash e : t^* \;;\; D_1 \cup D_2}$$

$$\text{where } \begin{cases} L = \{\ell_{i_1}, \ldots, \ell_{i_k}\} \\ e = \mathtt{case}\, e_1^*\, \mathtt{of} \\ \quad (f_{i_1}\ \mathtt{as}\ C_{i_1}(x)) \Rightarrow \mathit{apply}_{i_1}(f_{i_1}, e_2^*) \\ \quad \mid \ldots \\ \quad \mid (f_{i_k}\ \mathtt{as}\ C_{i_k}(x)) \Rightarrow \mathit{apply}_{i_k}(f_{i_k}, e_2^*) \end{cases}$$

The algorithm for defunctionalization works by recursion on the typing derivation of the source program. A typing derivation for the judgement $\psi \mid \Gamma \vdash e : s$ in the source language is translated into the target language derivation of $\psi^*, \Gamma^* \vdash e^* : s^* \,;\, D$, where

- For types, $\mathtt{int}^* = \mathtt{int}$ and $(s \to t, \{\ell_1, \ldots, \ell_k\})^* = \alpha_1 \cup \ldots \cup \alpha_k$.
- For label contexts,

$$(\ell_1 \mapsto (\Gamma_1, s_1, t_1), \ldots)^* = (\alpha_1, \ldots, C_1 : \Gamma_1^\dagger \to \alpha_1, \ldots, make_1 : \Gamma_1^\dagger \to \alpha_1, \ldots, \\ apply_1 : \alpha_1 \times s_1^* \to t_1^*, \ldots)$$

  where $(x_1 : s_1, \ldots, x_k : s_k)^\dagger = (s_1^* \times \ldots \times s_k^*)$. The ψ^* thus provide the specifications for the abstract types, application functions and creation functions (*i.e.*, the *make*'s) for every function label defined in the program. In an actual implementation, one might want to have *apply* functions defined for sets of labels that label the application sites in a program. However, for technical simplicity, we do not pursue this direction in this paper.
- For type contexts, $(x_1 : s_1, \ldots, x_k : s_k)^* = (x_1 : s_1^*, \ldots, x_k : s_k^*)$.
- For terms, e^* is a term in the target and D is a collection of declarations in the target. The declarations in the target provide implementations of the abstract types, application functions, and creation functions specified in ψ^*.

The complete specification appears in Table 6. Consider, for instance, the translation rule for applications. The algorithm uses the information encoded in ψ as well as in the context Γ. Note that $apply_1, \ldots, apply_k$ are all specified in ψ^* and defined in D_1 since $L \subseteq dom(\psi)$.

The translation seems straightforward, but it must be proven correct; that is, if we translate a source program, the source and target programs agree on the answers they produce. Correctness is particularly crucial here, since the proof of correctness of various transformations relies on the proof of correctness of defunctionalization. There are two parts to correctness. First, one must establish that the translation produces well-formed programs.

Lemma 1. *If $\psi \mid \Gamma \vdash e : s$ is provable in PCFA and $(\psi \mid \Gamma \vdash e : s) \Rightarrow (\psi^*, \Gamma^* \vdash e^* : s^* \,;\, D)$ then $\psi^*, \Gamma^* \vdash e^* : s^*$ is provable in the target language.*

A similar property holds for every declaration $d \in D$.

Theorem 4. *If $\psi \mid \emptyset \vdash e : \mathtt{int}$ is provable in PCFA and $(\psi \mid \emptyset \vdash e : \mathtt{int}) \Rightarrow (\psi^* \vdash e^* : \mathtt{int} \,;\, D)$ then $\emptyset \vdash \mathtt{let}\, D\, \mathtt{in}\, e^* : \mathtt{int}$ is provable in the target language.*

The proofs go by induction on the typing derivation in the source language.

A more difficult property to establish is correctness of the translation, which relates the final answers produced by a program and its translation.

Theorem 5 (Correctness). *If $\psi \mid \emptyset \vdash e : \mathtt{int}$ is provable in PCFA and $(\psi \mid \emptyset \vdash e : \mathtt{int}) \Rightarrow (\psi^* \vdash e^* : \mathtt{int} \,;\, D)$ then $(e, \emptyset) \Downarrow n$ in PCFA iff $(\mathtt{let}\, D\, \mathtt{in}\, e^*, \emptyset) \Downarrow n$ in the target language.*

Proof. (Sketch) Using the Adequacy Theorem 3, it is enough to show $(e, \emptyset) \Downarrow n$ iff $[\![\emptyset \vdash \mathtt{let}\, D\, \mathtt{in}\, e^* : \mathtt{int}]\!]\emptyset\emptyset = n$. Again, we use induction for the $(\Leftarrow)$ direction and lexicographic induction for the $(\Rightarrow)$ direction.

The reason for the lexicographic induction in the $(\Rightarrow)$ direction is due to recursion. Indeed, we know of no other proofs of correctness (that is, both directions) for ordinary or CFA-based defunctionalization in the presence of recursion.

5 Flow-based Inlining

The first example, flow-based inlining or what Shivers calls "super β-reduction", is best described by way of the example from the introduction. We want to show that

```
let fun^{ℓ1} g x = x in
let fun^{ℓ2} f h =
   (let fun^{ℓ3} u y =
    if0 y then (h @_{ℓ1} 1) else f @_{ℓ2} h @_{ℓ3} (h @_{ℓ1} y)
    in u) in
f @_{ℓ2} g @_{ℓ3} 2
```

is the same as

```
let fun^{ℓ1} g x = x in
let fun^{ℓ2} f h =
   (let fun^{ℓ3} u y =
    if0 y then (g @_{ℓ1} 1) else f @_{ℓ2} g @_{ℓ3} (g @_{ℓ1} y)
    in u) in
f @_{ℓ2} g @_{ℓ3} 2
```

where the example is slightly reformulated to take into account that recursive functions in PCFA take one argument. Our method of proof is to prove that the defunctionalized versions of these two programs are equivalent in the denotational model of the target. By the Adequacy Theorem 3, the programs in the target will return the same results operationally; thus, by the Correctness Theorem 5, the original programs in PCFA will return the same results operationally.

The translation of the original and alternative programs appears in Figure 2. The proof of equivalence of the above two pieces of code uses least fixpoint reasoning and beta reduction. For the first piece of code, we can β-reduce it to

```
apply3 (apply2(f,g), 2)
  = apply3 (make3 (g,f,g), 2)
  = apply3 (C3 <g,f,g>, 2)
  = if0 2 then apply1 (g, 1)
    else apply3 (apply2(f,g), apply1(g,2))
  = if0 2 then apply1 (g, 1)
    else apply3 (make3 (g,f,g), apply1(g,2))
```

and for the second piece of code, we can β-reduce it to

```
apply3 (apply2(f,g), 2)
  = apply3 (make3 (g,f,g), 2)
  = apply3 (C3 <g,f,g>, 2)
```

```
let tagtype a1 = C1 of unit
    fun make1 <> = C1<>
    fun apply1 (g as C1<>, x) = x
    tagtype a2 = C2 of a1
    fun make2 (g) = C2<g>
    fun apply2 (f as C2<g>, h) = let val u = make3(h, f, g) in u
    tagtype a3 = C3 of a1 x a2 x a1
    fun make3 (<h, f, g>) = C3 <h, f, g>
    fun apply3 (u as C3<h, f, g>, y) =
      if0 y then apply1 (h,1)
            else apply3 (apply2(f,h),apply1(h,y))
in let val g = make1 <> in
   let val f = make2 (g) in
   apply3 (apply2(f,g), 2)

let tagtype a1 = C1 of unit
    fun make1 <> = C1<>
    fun apply1 (g as C1<>, x) = x
    tagtype a2 = C2 of a1
    fun make2 (g) = C2<g>
    fun apply2 (f as C2<g>, h) = let val u = make3(h, f, g) in u
    tagtype a3 = C3 of a1 x a2 x a1
    fun make3 (<h, f, g>) = C3 <h, f, g>
    fun apply3 (u as C3<h, f, g>, y) =
      if0 y then apply1 (g,1)
            else apply3 (apply2(f,g),apply1(g,y))
in  let val g = make1 <> in
    let val f = make2 (g) in
    apply3 (apply2(f,g), 2)
```

Fig. 2. Translation of flow-based inlining examples.

```
= if0 2 then apply1 (g, 1)
  else apply3 (apply2(f,g), apply1(g,2))
= if0 2 then apply1 (g, 1)
  else apply3 (make3 (g,f,g), apply1(g,2))
```

Now a simple fixpoint induction on `apply3` suffices to prove the equivalence.

We can use our methods to prove general theorems about flow-based inlining. For example,

Theorem 6. *Suppose* $\psi \mid \emptyset \vdash e : \mathtt{int}$, *where e is the term*

$$\mathtt{let\ fun}^{\ell}\ f\ (x:s) = e_0\ \mathtt{in}\ K[y\ @_{\{\ell\}}\ e_1],$$

and K is a context (a term with a hole) that does not bind f. Let e' *be the term*

$$\mathtt{let\ fun}^{\ell}\ f\ (x:s) = e_0\ \mathtt{in}\ K[f\ @_{\{\ell\}}\ e_1].$$

Then $(e,\emptyset) \Downarrow n$ *iff* $(e',\emptyset) \Downarrow n$*.*

6 Lightweight Defunctionalization

Section 4 describes a relatively naive version of defunctionalization in which a function f in a source program is represented as a tagged tuple in the target. The tuple contains the values of all the free variables of the function. It is possible to be more clever. Among the set of free variables of a function f, if there is a subset S that is guaranteed to be in scope whenever f is applied, then S can be safely excluded from the tuple, provided for each $x \in S$, both f and x do not escape their respective scopes of definition.

The goal of this section is to develop this optimization, called "lightweight defunctionalization" in analogy with lightweight closure conversion [14, 39]. As with flow-based inlining, we prove the correctness of it using the correctness of ordinary defunctionalization. In contrast to flow-based inlining, the translation is *not* a source-to-source transformation; it is rather a source-to-target transformation, with a slightly modified type system for the source.

The possibilities for lightweight defunctionalization are best demonstrated by two examples. Example 1 shows when such an optimization is possible:

```
let val x = 1 in
let fun^ℓ1 g (y:int) = succ x in
g @_{ℓ1} 2
```

In this example, the naive representation of function `g` is the tagged tuple $C_1\langle \mathrm{x}\rangle$, since x is free in the body of `g`. Notice, though, that at the only application site for `g`, the variable `x` is always in scope. Therefore, it is not necessary to include it in the tagged tuple $C_1\langle \mathrm{x}\rangle$; the representation of `g` can just be $C_1\langle\rangle$.

Example 2 shows some of the subtlety in making such an optimization:

```
let fun^ℓ1 g(x:int) =
  let fun^ℓ2 u(y:(int→int,{ℓ3,ℓ4})) =
    let w = y @_{ℓ3,ℓ4} 3 in let fun^ℓ3 v(z:int) = x+w in v in u in
g @_{ℓ1} 0 @_{ℓ2} (g @_{ℓ1} 1 @_{ℓ2} (let fun^ℓ4 h(p:int) = 2 in h))
```

Here, the function g is first called with argument 1, and the function u is called with argument ℓ_4. The result of the application is the function ℓ_3, which is then fed back via a call to g that binds x to 0 and y to ℓ_3. Thus simultaneous bindings of x, one to 1 and the other to 0 are alive during the computation. Since the function ℓ_3 is one of the functions called at the application site (y $@_{\{\ell_3,\ell_4\}}$ 3), one might think that x need not be in the representation of ℓ_3, since x is always in scope. This is, however, wrong: the function ℓ_3 escapes the scope of its definition and gets bound to y with a different binding for its free variable x.

These examples give us clues regarding the specification of lightweight defunctionalization. It is helpful to think how we might translate a source term into the target language. Suppose a subexpression e in the source has type t in environment Γ. Suppose also that among the function labels occurring in e, there is a subset L that occurs neither in e's result type t nor in the environment Γ. Then we can conclude that these functions will not escape from e, will not be used any further in the computation, and are hence *local* to e. Therefore, in e's translation, we should make sure that the corresponding *make* and *apply* functions for each label in L are also local, since these *make*'s and *apply*'s will never be used in the rest of the computation. This notion of locality is similar to locality in the region calculus [40].

With this intuition, one can think of the *make* and *apply* functions as capabilities that are currently available for use by e. When we make some of them local, the rest of the program cannot use these capabilities any more. This change of perspective dictates a small change in the reading of typing judgements in PCFA. Before, we used the context ψ in $\psi \mid \Gamma \vdash e : s$ to determine the labels of functions, and their attendant types, when typechecking e. Now, we will use ψ like a typing context, and remove capabilities from it when we wish to make functions local to a particular scope.

The changes to the type system for PCFA appropriate for lightweight defunctionalization appear in Table 7; we call the new language LPCFA. One rule is modified, namely the rule for function declarations, and one rule is added. The function declaration rule makes clear that ψ is now slightly different: it maps a label ℓ to a quadruple (Δ, Π, s, t). In the quadruple, Δ is the global environment, *i.e.*, free variables in ℓ that are always in scope and which therefore need not appear in the representation of the function, and Π is the local environment of free variables that may escape their scopes of definition and which therefore must appear in the representation of the function in the target. Algorithmically, the split between global and local variables would have to be determined; in the type system, the "guess" of the split is merely checked.

We reconsider the examples in turn. In Example 1, the type of x is int, the type of g is $(\mathtt{int} \to \mathtt{int}, \{\ell_1\})$ and the type of the whole program is int. Whenever g is applied, the variable x is in its scope. The typing derivation uses the new rules to make this locality explicit. First, when typechecking the bindings of the inner let, we need an appropriate ψ to check

$$J_1 = \psi \mid x : \mathtt{int} \vdash \mathtt{fun}^{\ell_1}\, g\,(y : \mathtt{int}) = \mathtt{succ}\, x \rhd (g : \mathtt{int} \to \mathtt{int}, \{\ell_1\})$$

To make x global, we choose

$$\psi = [\ell_1 \mapsto (\{x : \mathtt{int}\}, \emptyset, \mathtt{int}, \mathtt{int})]$$

Table 7. Modified type rules for LPCFA.

Well-formed declarations $\boxed{\psi \mid \Gamma \vdash D \triangleright \Gamma'}$

$$\frac{\psi \mid \Gamma, f:(s \to t, \{\ell\}), x:s \vdash e:t \quad \psi(\ell) = (\Delta, \Pi, s, t) \quad (\Delta, \Pi) = \Gamma}{\psi \mid \Gamma \vdash (\mathtt{fun}^{\ell}\, f(x:s) = e) \triangleright (f : s \to t, \{\ell\})}$$

Well-formed expressions $\boxed{\psi \mid \Gamma \vdash e : s}$

$$\frac{\psi, \ell_1 : (\Gamma, \Pi_1, t_1, t_1') \ldots \ell_k : (\Gamma, \Pi_k, t_k, t_k') \mid \Gamma \vdash e : s}{\psi \mid \Gamma \vdash e : s}$$

$$\text{where } \begin{cases} \ell_1 \ldots \ell_k \text{ occurs in } e \\ \ell_1 \ldots \ell_k \text{ does not occur in } \mathit{Labels}(s, \Gamma) \\ \ell_1 \ldots \ell_k \notin \mathit{dom}(\psi) \end{cases}$$

It is then evident that the judgement J_1 is provable, and from that it is easy to see that

$$\psi \mid x : \mathtt{int} \vdash (\mathtt{let\ fun}^{\ell_1}\, g(y : \mathtt{int}) = \mathtt{succ}\, x \;\mathtt{in}\; g \,@_{\{\ell_1\}}\, 2) : \mathtt{int}$$

is provable. Now we can make g local, since ℓ_1 does not appear either in the result type $\mathtt{int}$, or in the type context $x : \mathtt{int}$. Thus,

$$\emptyset \mid x : \mathtt{int} \vdash (\mathtt{let\ fun}^{\ell_1}\, g(y : \mathtt{int}) = \mathtt{succ}\, x \;\mathtt{in}\; g \,@_{\{\ell_1\}}\, 2) : \mathtt{int}$$

is provable.

The use of the new rule for locality is invisible: it does not have a syntactic marker in the source language. Nevertheless, it is crucial in the translation to the target. The modifications to the translation appear in Table 8, where the translation of label contexts is now determined by

$$\begin{aligned}(\ell_1 \mapsto (\Delta_1, \Pi_1, s_1, t_1), \ldots, \ell_n \mapsto (\Delta_n, \Pi_n, s_n, t_n))^* = \\ \alpha_1, C_1 : \Pi_1^\dagger \to \alpha_1, \ldots, \alpha_n, C_n : \Pi_n^\dagger \to \alpha_n, \ldots, \\ \mathit{make}_1 : \Pi_1^\dagger \to \alpha_1, \mathit{apply}_1 : \alpha_1 \times s_1^* \to t_1^*, \ldots, \\ \mathit{make}_n : \Pi_n^\dagger \to \alpha_n, \mathit{apply}_n : \alpha_n \times s_n^* \to t_n^*\end{aligned}$$

The rest of the translation follows *mutatis mutandis*. Note how the type of make_i is $\Pi_i^\dagger \to \alpha_i$, and that Π_i does not necessarily include *all* of the free variables of the recursive function. This fact is used in the translation of the new rule. In this case, we know that $\ell_1, \ldots, \ell_k$ are all the function labels that can be made local in the source term e. Accordingly, in the translation, the definitions of the tagged types $\alpha_1, \ldots, \alpha_k$, and the functions $\mathit{make}_1, \ldots, \mathit{make}_k$ and $\mathit{apply}_1, \ldots, \mathit{apply}_k$ are made local to e^*. Moreover, these local definitions are subtracted from D.

Table 8. Modified rules for lightweight defunctionalization of LPCFA.

$$\frac{\psi \mid \Gamma, f : (s \to t, \{\ell\}), x : s \vdash e : t \quad \psi(\ell) = (\Delta, \Pi, s, t) \quad (\Delta, \Pi) = \Gamma}{\psi \mid \Gamma \vdash (\mathtt{fun}^{\ell} f\,(x : s) = e) \rhd (f : s \to t, \{\ell\})} \quad \Rightarrow$$

$$\frac{\psi^*, \Gamma^*, f : \alpha, x : s^* \vdash e^* : t^* \;;\; D}{\psi^*, \Gamma^* \vdash \mathit{make}\langle y_1, \ldots, y_m\rangle : \alpha \;;\; D \cup D'}$$

$$\text{where} \begin{cases} \Pi = \langle y_1 : u_1, \ldots, y_m : u_m\rangle \\ D' = \{\, \mathtt{tagtype}\,\alpha \;=\; C\;\mathtt{of}\;\Pi^*, \\ \qquad \mathtt{fun}\;\mathit{make}\;(\langle y_1, \ldots, y_m\rangle) = C\langle y_1, \ldots, y_m\rangle, \\ \qquad \mathtt{fun}\;\mathit{apply}\;(f\;\mathtt{as}\;C\langle y_1, \ldots, y_m\rangle, x) = e^* \} \end{cases}$$

$$\frac{\psi, \ell_1 : (\Gamma, \Pi_1, t_1, t_1') \ldots \ell_k : (\Gamma, \Pi_k, t_k, t_k') \mid \Gamma \vdash e : s}{\psi \mid \Gamma \vdash e : s} \quad \Rightarrow$$

$$\frac{\psi^*, \alpha_j,\, C_j,\, \mathit{make}_j,\, \mathit{apply}_j,\, \Gamma^* \vdash e^* : s^* \;;\; D \quad j = 1 \ldots k}{\begin{array}{l} \psi^*, \Gamma^* \vdash \mathtt{let}\;\mathtt{tagtype}\,\alpha_j \;=\; C_j\;\mathtt{of}\;\Pi_j^* \\ \qquad \mathtt{fun}\;\mathit{make}_j\;(\langle z_{1j}, \ldots, z_{p_j}\rangle) = C_j\langle z_{1j}, \ldots, z_{p_j}\rangle \\ \qquad \mathtt{fun}\;\mathit{apply}_j\;(f_j\;\mathtt{as}\;C_j\langle z_{1j}, \ldots, z_{p_j}\rangle, u : t_j^*) = e_j^* \\ \quad \mathtt{in}\;\; e^* : s^* \;;\; D - D' \end{array}}$$

$$\text{where} \begin{cases} \ell_1 \ldots \ell_k \text{ occurs in } e \\ \ell_1 \ldots \ell_k \;\notin\; \mathit{Labels}(s, \Gamma) \\ \ell_1 \ldots \ell_k \notin \mathit{dom}(\psi) \\ \Pi_j = z_{1j} : s_{1j}, \ldots, z_{p_j} : s_{p_j} \\ C_j : \Pi_j^* \to \alpha_j,\; \mathit{make}_j : \Pi_j^* \to \alpha_j,\; \mathit{apply}_j : \alpha_j \times t_j^* \to (t_j')^* \\ D' = \{\, \mathtt{tagtype}\,\alpha_j \;=\; C_j\;\mathtt{of}\;\Pi_j^* \\ \qquad \mathtt{fun}\;\mathit{make}_j(\ldots) = \ldots, \\ \qquad \mathtt{fun}\;\mathit{apply}_j(\ldots) = \ldots \mid j = 1 \ldots k\} \end{cases}$$

It is helpful to go back to Example 1 to see how the translation works. Note first that

$$\psi^* = \alpha_1, C_1 : \texttt{unit} \to \alpha_1, \mathit{make}_1 : \texttt{unit} \to \alpha_1,\ \mathit{apply}_1 : \alpha_1 \times \texttt{int} \to \texttt{int}.$$

The translation of the judgement J_1 is then

$$\begin{array}{l} \psi^*, x : \texttt{int} \vdash \texttt{let val}\ g = \mathit{make}_1 \langle\rangle\ \texttt{in} \\ \qquad\qquad \texttt{case}\ g\ \texttt{of}\ (g\ \texttt{as}\ C_1\ \langle\rangle)\ \Rightarrow\ \mathit{apply}_1(g, 2) : \texttt{int}\ ; D \end{array}$$

where

$$\begin{array}{l} D = \{\, \texttt{tagtype}\, \alpha_1\ =\ C_1\ \texttt{of unit},\ \texttt{fun}\ \mathit{make}_1\ (\langle\rangle) = C_1 \langle\rangle, \\ \qquad \texttt{fun}\ \mathit{apply}_1\ (g\ \texttt{as}\ C_1 \langle\rangle,\ y) = \texttt{succ}\ x\}. \end{array}$$

An application of the translation rules yields the following program (call it P):

$$\begin{array}{l} \emptyset \vdash \texttt{let val}\ x = 1\ \texttt{in} \\ \quad \texttt{let tagtype}\, \alpha_1\ =\ C_1\ \texttt{of unit} \\ \qquad \texttt{fun}\ \mathit{make}_1\ (\langle\rangle) = C_1 \langle\rangle \\ \qquad \texttt{fun}\ \mathit{apply}_1\ (g\ \texttt{as}\ C_1 \langle\rangle,\ y) = \texttt{succ}\ x\ \texttt{in} \\ \quad \texttt{let val}\ g = \mathit{make}_1 \langle\rangle\ \texttt{in} \\ \quad \texttt{case}\ g\ \texttt{of}\ (g\ \texttt{as}\ C_1\ \langle\rangle)\ \Rightarrow\ \mathit{apply}_1(g, 2) : \texttt{int}\ ; \emptyset \end{array}$$

which is the translation of Example 1.

In Example 2, we need an appropriate ψ to check

$$J_2 = \psi \mid \emptyset \vdash \texttt{fun}^{\ell_1}\, g\, (x : \texttt{int}) = \ldots \rhd (g : s)$$

where $s = (\texttt{int} \to ((\texttt{int} \to \texttt{int}, \{\ell_3, \ell_4\}) \to (\texttt{int} \to \texttt{int}, \{\ell_3\})), \{\ell_2\})$. Then we can show

$$J_3 = \psi \mid g : s \vdash g\ @_{\{\ell_1\}}\ 0\ @_{\{\ell_2\}}\ (g\ @_{\{\ell_1\}}\ 1\ @_{\{\ell_2\}}\ e) : t$$

where $t = (\texttt{int} \to \texttt{int}, \{\ell_3\})$ and $e = (\texttt{let fun}^{\ell_4}\, h\, (p : \texttt{int}) = 2\ \texttt{in}\ h)$. But what can ψ contain? Clearly, from J_3 it must contain bindings for ℓ_1, ℓ_2 and ℓ_4, otherwise the applications cannot take place. Moreover, to typecheck g's body, we need to typecheck the judgement

$$J_4 = \psi \mid g : s, x : \texttt{int}, u : r \vdash (\texttt{let}\ w = y\ @_{\{\ell_3, \ell_4\}}\ 3\ \texttt{in}\ e') : p$$

where $r = ((\texttt{int} \to \texttt{int}, \{\ell_3, \ell_4\}) \to (\texttt{int} \to \texttt{int}, \{\ell_3, \ell_4\}), \{\ell_2\})$ and $p = (\texttt{int} \to \texttt{int}, \{\ell_3\})$ and $e' = (\texttt{let fun}^{\ell_3}\, v\, (z : \texttt{int}) = x + w\ \texttt{in}\ v)$. To typecheck the application in J_4, ψ must contain a binding for ℓ_3. Finally, note that the return type p indicates that ℓ_3 escapes the scope of its definition. So ℓ_3 cannot be made local. The defunctionalized version of Example 2 and its lightweight defunctionalized version are exactly the same and is shown below. Note how the *make* and *apply* functions for each of the labels are available globally.

```
let tagtype α₁ = C₁ of unit
    tagtype α₂ = C₂ of α₁×int
    tagtype α₃ = C₃ of α₁×int×α₂×(α₁∪α₃)×int
    tagtype α₄ = C₄ of α₁
    fun make₁() = C₁⟨⟩
    fun make₂(d,x) = C₂⟨d,x⟩
    fun make₃(d,x,e,v,w) = C₃⟨d,x,e,v,w⟩
    fun make₄(d) = C₄⟨d⟩
    fun apply₁(g as C₁⟨⟩, x:int) = let val u = make₂(g,x) in u
    fun apply₂(u as C₂⟨d,x⟩, y) =
      let val w =
        (case y of (v as C₃⟨d',x',e',v',w'⟩) => apply₃(v,3)
                 | (h as C₄⟨d⟩) => apply₄(h,3)) in
      let val v = make₃(x,w) in v
    fun apply₃(v as C₃⟨d,x,e,v,w⟩,z) = x + w
    fun apply₄(h as C₄⟨d⟩,p) = 2
in  let val g = make₁⟨⟩ in
    apply₂(apply₁(g,0),apply₂(apply₁(g,1),
           let val h = make₄(g) in h))
```

We use principles from the theory of data abstraction to show that the program produces the same output as the original defunctionalized program. Our strategy is to produce an intermediate transformation of the program, where the transformation is essentially defunctionalization, but takes into account the new rule. This allows us to use the Fundamental Theorem of Logical Relations [24, 33] to relate two programs with similar scoping structure. Finally, we can show that the program produced by the intermediate transformation and the original defunctionalized program are β-convertible.

Consider the use of ordinary defunctionalization on Example 1, yielding:

$$
\begin{array}{l}
\emptyset \vdash \texttt{let tagtype}\,\alpha_1 \;=\; C_1\,\texttt{of int} \\
\qquad\qquad \texttt{fun}\,\mathit{make}_1\;(x) = C_1\langle x\rangle \\
\qquad\qquad \texttt{fun}\,\mathit{apply}_1\;(g\;\texttt{as}\;C_1\langle x\rangle,\,y) = \texttt{succ}\,x\;\texttt{in} \\
\qquad \texttt{let val}\,x = 1\;\texttt{in} \\
\qquad \texttt{let val}\,g = \mathit{make}_1(x)\;\texttt{in} \\
\qquad \texttt{case}\,g\,\texttt{of}\,(g\;\texttt{as}\;C_1\;\langle x\rangle)\;\Rightarrow\;\mathit{apply}_1(g,2):\texttt{int}\;;\,D.
\end{array}
$$

We can also choose to defunctionalize by taking advantage of the new locality rule, but without reducing the size of the environments, *i.e.*, C_1 still carries an `int`. Under this version, the translated program is

$$
\begin{array}{l}
\emptyset \vdash \texttt{let val}\,x = 1\;\texttt{in} \\
\qquad \texttt{let tagtype}\,\alpha_1 \;=\; C_1\,\texttt{of int} \\
\qquad\qquad \texttt{fun}\,\mathit{make}_1\;(\langle\rangle) = C_1\langle x\rangle \\
\qquad\qquad \texttt{fun}\,\mathit{apply}_1\;(g\;\texttt{as}\;C_1\langle x\rangle,\,y) = \texttt{succ}\,x\;\texttt{in} \\
\qquad \texttt{let val}\,g = \mathit{make}_1\langle\rangle\;\texttt{in} \\
\qquad \texttt{case}\,g\,\texttt{of}\,(g\;\texttt{as}\;C_1\langle x\rangle)\;\Rightarrow\;\mathit{apply}_1(g,2):\texttt{int}\;;\,\emptyset
\end{array}
$$

It is simple to prove that these two programs are the same, using β-conversion, which is valid in the target language. To show that this last version is equivalent to the result of lightweight defunctionalization, (*i.e.*, program P), we use the Fundamental Theorem of Logical Relations, appropriately extended to the target language. We build a relation between the representations of the abstract type α_1, namely $R_{\alpha_1} = \{\langle C_1\langle\rangle, C_1(1)\rangle\}$. We then show that the implementations of $make_1$ and $apply_1$ are related in the scope where `x` is bound to 1, and thus the bodies of the two implementations are related. Since the relation at base type `int` is the equality relation, both programs produce the same final answers.

More formally, we can prove:

Theorem 7 (Correctness). *If* $\psi \mid \emptyset \vdash e : \mathtt{int}$ *is provable in LPCFA and* $(\psi \mid \emptyset \vdash e : \mathtt{int}) \Rightarrow (\psi^* \vdash e^* : \mathtt{int}\ ;\ D)$ *via lightweight defunctionalization, then* $(e, \emptyset) \Downarrow n$ *in LPCFA iff* $(\mathtt{let}\ D\ \mathtt{in}\ e^*, \emptyset) \Downarrow n$ *in the target language.*

7 Discussion

We have shown how to prove the correctness of ordinary CFA-based defunctionalization using denotational methods, and how to prove the correctness of two CFA-based transformations using this result. Our methods seem to be more modular and reusable than previous correctness proofs. Nevertheless, the proof methods developed here are not a silver bullet: although the proof of correctness for defunctionalization can serve as a building block, the proofs of correctness for the transformations must be carried out afresh.

Defunctionalization, first formulated by Reynolds, in his celebrated paper on definitional interpreters [32], is an important program transformation technique that has surfaced in several contexts. Reynolds developed defunctionalization as an aid to understanding meta-circular interpreters for higher-order applicative languages since such interpreters do not "shed much light on the nature of higher-order functions"; an interpreter for a higher-order applicative language must thus be written in a *first-order* metalanguage. Defunctionalization is proposed as a way of systematically eliminating higher-order functions by representing them using records containing the free variables of these functions: these records are essentially closures. More recently, CFA-based defunctionalization has been used in whole-program compilers for ML [7, 41] for efficient code generation. In a recent dissertation, Boquist shows how to transform Haskell programs into efficient, first-order programs [6]. His method uses (essentially) a form of CFA to reduce the complexity of the defunctionalized program. However, no formal proofs of correctness for the transformations or of defunctionalization are provided. Finally, Wang and Appel in their work on type-safe garbage collection [45] assume whole-program compilation and CPS convert and defunctionalize source programs before performing garbage collection.

Defunctionalization plays two distinct roles in this paper: as a basis for proofs of correctness, and as a basis for program transformations. The two roles come together here in the proof of correctness of lightweight defunctionalization.

We have sketched, but not reported here, a CFA-based defunctionalization for two other versions of PCFA, one untyped and one with recursive types. Interestingly, neither change in the source seems to require additional features in the target. In the Appendix we sketch an extension of PCFA with imperative features, namely, references, assignment and dereference. For this language, CFA also calculates, in addition to the possible set of functions, the possible set of references that may be the result of evaluation at a program point. In this case, however, the target language needs to be augmented with imperative features. To make references local, a rule similar to the "locality" rule in LPCFA may be employed. Going further, one might consider a source language with simple concurrency primitives. We conjecture that this would not add higher-order features to the target language, and thus avoid powerdomains in the denotational semantics of the target.

A fundamental intuition in this paper is that control-flow analysis allows one to choose different concrete representations of data (*e.g.*, functions, references, *etc*.) This is why it can be used as the basis of program transformations. The choice is made explicit in the translation of *e.g.*, a function type (or a reference type) in the source into an *abstract type* in the target. Then, creation of a function and application of a function become operations on the abstract type. Different program transformations can choose to implement a function differently by choosing different concrete representations for it. This view allows one to use common semantic techniques (*e.g.*, principles of data abstraction) to prove correctness of the transformations. The intuition of using CFA to choose representations does not seem to be explicit in Steckler and Wand's treatment of lightweight closure conversion [39].

Defunctionalization, as reported in this paper, is a whole-program transformation. It crucially depends on results of a whole-program CFA. But what happens if we want to compile separate program modules independently? We are currently developing an approach based on the restricted introduction of variables ξ ranging over sets of labels. Corresponding to these variables, we introduce a new class of types $(s \to s, \xi)$. These types are used to describe a module's imported functions. In the defunctionalization of a module, an imported function f maps into a representation of f and any apply functions necessary to apply f and its "residuals". For example, if f has type $((\mathtt{int} \to \mathtt{int}, \xi) \to \mathtt{int}, \xi')$, we need apply functions $apply_{\xi}$ and $apply_{\xi'}$.

To formalize these notions we need a concrete model of linking. Since the essence of defunctionalization is the transformation of higher-order programs to first-order ones, we want a linking model where linked programs are first-order programs. One immediate complication is that non-recursive modules map to recursive modules under defunctionalization. To see this, suppose that module M imports a function f from module M_f. Now, suppose that (in M_f) the definition of f takes a function as argument and applies this function within its body (i.e. this application occurs in M_f). Also, suppose that in M, the function f is applied to a locally defined function h. Then we have (a) M must import a representation of f and "$apply_f$" from M_f, and (b) M_f must import "$apply_h$" from M since h is applied within M_f. We remark that the type system must be restricted in a number of ways to ensure that the linking can be done statically. We are working on a system based on existential and universal types to capture this recursive linking process.

Another potential example for using defunctionalization is useless variable elimination [38, 44], although in this case Kobayashi [22] has already given an elegant operational proof of correctness. We also conjecture that our earlier work on dependency analyses [1] can be extended to handle useless variable elimination. One might apply our techniques to study access-control security, as in the SLam calculus [16], whose type-based specification is closely related to CFA.

Acknowledgements

Anindya Banerjee is a member of the Church Project, and was supported in part by NSF grant EIA-9806835.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL'99*, ACM Press.
2. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., 1986.
3. A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
4. A. Banerjee. A modular, polyvariant and type-based closure analysis. In *ICFP'97*, ACM Press.
5. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North-Holland, 1981. Revised Edition, 1984.
6. U. Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Göteborg University, 1999.
7. H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In *ESOP 2000*, number 1782 in Lect. Notes in Computer Sci. Springer-Verlag.
8. H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages (Extended Summary). Available from URL:http://www.neci.nj.nec.com/homepages/jagannathan/publications.html, 2000.
9. J. Dean, G. DeFouw, D. Grove, and C. Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *OOPSLA'96*, ACM Press.
10. A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *ICFP'97*, ACM Press.
11. M. Fähndrich and A. Aiken. Making set-constraint based program analyses scale. Technical Report 96-917, University of California, Berkeley, Computer Science Division, 1996.
12. C. Flanagan and M. Felleisen. Componential set-based analysis. In *ACM Trans. on Programming Languages and Systems*, 21(2):370-416, March 1999.
13. C. A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.
14. J. Hannan. Type systems for closure conversion. In H. R. Nielson and K. L. Solberg, editors, *Proceedings of the Workshop on Types in Program Analysis*. Technical Report PB-493, University of Aarhus, 1995.
15. N. Heintze. Control-flow analysis and type systems. In *SAS'95*, number 983 in Lect. Notes in Computer Sci. Springer-Verlag.
16. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *POPL'98*, ACM Press.
17. U. Hölzle and O. Agesen. Dynamic vs. static optimization techniques for object-oriented languages. *Theory and Practice of Object Systems*, 1(3), 1996.
18. S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL'95*, ACM Press.

19. S. Jagannathan and A. Wright. Effective flow analysis for avoiding runtime checks. In *SAS'95*, number 983 in Lect. Notes in Computer Sci. Springer-Verlag.
20. N. D. Jones. Flow analysis of lambda expressions (preliminary version). In S. Even and O. Kariv, editors, *Automata, Languages and Programming: Eighth Colloquium*, volume 115 of *Lect. Notes in Computer Sci.*, pages 114–128. Springer-Verlag, July 1981.
21. G. Kahn. Natural semantics. In *Proceedings Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lect. Notes in Computer Sci.*, New York, 1987. Springer-Verlag.
22. N. Kobayashi. Type-based useless variable elimination. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 84–93, ACM, 2000.
23. Y. Minamide, G. Morrisett and R. Harper. Typed Closure Conversion. In *POPL'96*, ACM Press.
24. J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
25. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
26. L. R. Nielsen. A Denotational Investigation of Defunctionalization. *Progress Report, BRICS Ph.D. School*, June 1998.
27. F. Nielson. Program transformations in a denotational setting. *ACM Trans. on Programming Languages and Systems*, 7(3):359–379, 1985.
28. F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL'97*, ACM Press.
29. F. Nielson, H. R. Nielson and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
30. J. Palsberg. Closure analysis in constraint form. *ACM Trans. on Programming Languages and Systems*, 17(1):47-62, 1995.
31. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.
32. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Meeting (Boston, 1972)*, pages 717–740, 1972.
33. J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North Holland, Amsterdam, 1983.
34. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998.
35. J. C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, December 1998.
36. D. Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. *Theoretical Computer Science*, 121:411–440. Published version of unpublished manuscript, Oxford University, 1969.
37. P. Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988.
38. O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Available as Technical Report CMU-CS-91-145.
39. P. A. Steckler and M. Wand. Lightweight closure conversion. *ACM Trans. Programming Languages and Systems*, 19(1):48–86, January 1997.
40. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
41. A. Tolmach and D. P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.
42. M. Wand. Specifying the Correctness of Binding-Time Analysis. In *Journal of Functional Programming*, 3(3):365–387, July 1993.
43. M. Wand and W. Clinger. Set Constraints for Destructive Array Update Optimization. In *Journal of Functional Programming*, 2000, to appear.

44. M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *POPL'99*, ACM Press.
45. D. Wang and A. Appel. Type-preserving garbage collectors. In *POPL'01*, ACM Press.
46. K. Yi. An abstract interpretation for estimating uncaught exceptions in Standard ML programs. In *Science of Computer Programming*, 31(1):147-173, 1998.

A Imperative features

In this section we sketch how to extend the source language with imperative features: creation of reference cells, assignment, dereference. Control-flow analysis, in addition to calculating the set of possible function labels that can arise as a result of evaluation at a program point, also calculates the set of possible reference cells that can be the result of evaluation at a program point.

A.1 The source language

We show the extension of PCFA with imperative features below. Reference cells are annotated by the label of the cell. Dereference and assignments are labeled by the set of possible reference cells that maybe dereferenced or assigned at the particular program point. Finally, similar to function types in PCFA types of reference cells are also labelled.

$$\begin{aligned} s &::= \ldots \mid (\texttt{ref}\ s, L) \\ e &::= \ldots \mid \texttt{ref}^{\ell}\ e \mid \texttt{deref}_L\ e \mid \texttt{assign}_L\ e_1\ e_2 \end{aligned}$$

The type rules below provide a specification for CFA in the presence of references. The label context ψ now also contains labels of reference cells in its domain. Let ℓ be the label of a reference cell that can store a value of type s. Then $\psi(\ell) = s$.

$$\frac{\psi \mid \Gamma \vdash e : s}{\psi \mid \Gamma \vdash \texttt{ref}^{\ell}\ e : (\texttt{ref}\ s, \{\ell\})}\ \psi(\ell) = s$$

$$\frac{\psi \mid \Gamma \vdash e : (\texttt{ref}\ s, L)}{\psi \mid \Gamma \vdash \texttt{deref}_L\ e : s}\ L \subseteq dom(\psi)$$

$$\frac{\psi \mid \Gamma \vdash e_1 : (\texttt{ref}\ s, L) \quad \psi \mid \Gamma \vdash e_2 : s}{\psi \mid \Gamma \vdash \texttt{assign}_L\ e_1\ e_2 : s}\ L \subseteq dom(\psi)$$

Note how for the dereference and assignment rules, we have a similar side condition as for the application rule: a reference cell can be dereferenced or assigned to only if it is currently available in ψ.

A.2 The target language

The target language needs to be extended with imperative features also. The extension is standard: reference cells, dereference and assignment are added without any labels.

Type rules are as follows.

$$[Ref] \quad \frac{\Delta \vdash e : s}{\Delta \vdash \texttt{ref}\, e : \texttt{ref}\, s} \qquad [Deref] \frac{\Delta \vdash e : \texttt{ref}\, s}{\Delta \vdash \texttt{deref}\, e : s}$$

$$[Assign] \frac{\Delta \vdash e_1 : \texttt{ref}\, s \quad \Delta \vdash e_2 : s}{\Delta \vdash \texttt{assign}\, e_1\, e_2 : s}$$

Denotational semantics The only addition is the meaning of the reference type. Let $Loc = \mathbf{N}$. For a type environment η, we define

$$[\![\Delta \vdash \texttt{ref}\, s]\!]\eta \;=\; Loc.$$

A *state* over η is a function $(\mathbf{N} \to Storable_\eta) \times \mathbf{N}$, where

$$Storable_\eta \;=\; \{\langle\rangle\} + \mathbf{N} + Loc + \bigcup_{\alpha \in dom(\eta)} \eta(\alpha) + (Storable_\eta \times Storable_\eta)$$

Meanings of terms are now defined over an environment ρ and a state σ. For example

$$[\![\Delta \vdash \texttt{ref}\, e : \texttt{ref}\, s]\!]\eta\rho\sigma \;=\; (v, \sigma')$$

where

$$\begin{array}{ll} (v, \sigma_1) & = [\![\Delta \vdash e : s]\!]\eta\rho\sigma \\ \sigma_1 & = (f, n) \\ f'(x) & = \begin{cases} v, & \text{if } x = n+1 \\ f(x) & \text{otherwise} \end{cases} \\ \sigma' & = (f', n+1) \end{array}$$

A.3 Translation into target

- Type translation is extended as follows: Let $L = (\ell_1, \ldots, \ell_k)$. Then $(\texttt{ref}\, s, L)^* \;=\; \alpha_1 \cup \ldots \cup \alpha_k$.
- Label context translation is extended as follows:

$$\begin{array}{l} (\ell \mapsto s, \ldots)^* = (\alpha_\ell, C_\ell : (\texttt{ref}\, s^*) \to \alpha_\ell,\; makeref_\ell : s^* \to \alpha_\ell, \\ \qquad\qquad deref_\ell : \alpha_\ell \to s^* \;\; assign_\ell : \alpha_\ell \times s^* \to s^*, \ldots) \end{array}$$

We now show how judgements in the extended PCFA can be translated into the extended target:

$$\frac{\psi \mid \Gamma, e : s}{\psi \mid \Gamma \vdash \texttt{ref}^{\ell}\, e : (\texttt{ref}\, s, \{\ell\})}\ \psi(\ell) = s \qquad \Rightarrow$$

$$\frac{\psi^*, \Gamma^* \vdash e^* : s^* \ ;\ D}{\psi^*, \Gamma^* \vdash \mathit{makeref}_{\ell} e^* : \alpha_{\ell} \ ;\ D \cup D'} \quad \begin{cases} D' = \{\, \texttt{tagtype}\, \alpha_{\ell} \ = \ C_{\ell}\ \texttt{of}\ (\texttt{ref}\, s^*), \\ \qquad \texttt{fun}\ \mathit{makeref}_{\ell}\ (x) = C_{\ell}(\texttt{ref}\, x) \\ \qquad \texttt{fun}\ \mathit{deref}_{\ell}\ (C_{\ell}(x)) = \texttt{deref}(x) \\ \qquad \texttt{fun}\ \mathit{assign}_{\ell}\ (C_{\ell}(x), y) = \texttt{assign}\, x\, y\} \end{cases}$$

$$\frac{\psi \mid \Gamma \vdash e : (\texttt{ref}\, s, L)}{\psi \mid \Gamma \vdash \texttt{deref}_L\, e : s}\ L = \{\ell_1, \ldots, \ell_k\} \subseteq \mathit{dom}(\psi) \qquad \Rightarrow$$

$$\frac{\psi^*, \Gamma^* \vdash e^* : \alpha_1 \cup \ldots \cup \alpha_k \ ;\ D}{\psi^*, \Gamma^* \vdash \texttt{case}\ e^*\ \texttt{of}\ C_1(r) \Rightarrow \mathit{deref}_{\ell_1}(r) \mid \ldots \mid C_k(r) \Rightarrow \mathit{deref}_{\ell_k}(r) : s^* \ ;\ D}$$

$$\frac{\psi \mid \Gamma \vdash e_1 : (\texttt{ref}\, s, L) \quad \psi \mid \Gamma \vdash e_2 : s}{\psi \mid \Gamma \vdash \texttt{assign}_L\, e_1\, e_2 : s}\ L = \{\ell_1, \ldots, \ell_k\} \subseteq \mathit{dom}(\psi) \Rightarrow$$

$$\frac{\psi^*, \Gamma^* \vdash e_1^* : \alpha_1 \cup \ldots \cup \alpha_k \ ;\ D_1 \quad \psi^*, \Gamma^* \vdash e_2^* : s^* \ ;\ D_2}{\psi^*, \Gamma^* \vdash \texttt{case}\ e_1^*\ \texttt{of}\ C_1(r) \Rightarrow \mathit{assign}_{\ell_1}(C_1(r), e_2^*) \mid \ldots \mid C_k(r) \Rightarrow \mathit{assign}_{\ell_k}(C_k(r), e_2^* : s^* \ ;\ D_1 \cup D_2}$$