Inferring Annotated Types for Inter-procedural Register Allocation with Constructor Flattening

Torben Amtoft*†‡§
Kansas State University
tamtoft@cis.ksu.edu

Robert Muller†‡¶‖
Boston College
muller@cs.bc.edu

ABSTRACT

We introduce an annotated type system for a compiler intermediate language. The type system is designed to support inter-procedural register allocation and the representation of tuples and variants directly in the register file. We present an algorithm that generates constraints for assigning annotations, and prove its soundness with respect to the type system.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms

Languages, Reliability, Verification

Keywords

Type Systems, Effects, Register Allocation, Defunctionalization, Certifying Compilers

1. INTRODUCTION

The difficulty of implementing higher-order programming languages in a safe and efficient way is well-known. The combination of lexical scoping and function-values usually leads the compiler writer to represent functions as first-order closure data structures. These structures are usually stored in the heap so closure creation and application are expensive off-chip operations. This problem is compounded by the fact that such languages give rise to a programming style with very heavy reliance on function application. Thus, the expensive record packing and unpacking operations tend to occur frequently and register context information is frequently saved and restored around closure invocations.

Compiler writers naturally look to minimize these costs by computing more efficient function representations. But this introduces additional complexity in the compiler and is therefore more prone to error.

In this paper we introduce a type system that is designed to support the safe and efficient implementation of functions in ML-like languages. The type system is an effect-based system [13, 14] that supports inter-procedural register allocation and representations of tuples and variants directly in the register file. Our application of type systems for this purpose is closely related to the system developed by Johan Agat in [2, 1]. See section 5 for a comparison.

We also present an algorithm for inferring storage annotations. Our algorithm accepts as input a typed but unannotated program *up* that is

- *monomorphized*, that is function definitions have been copied so that each copy is invoked with one type only;
- *defunctionalized*, that is all functions are explicitly represented by closures;
- in *named form*, that is all intermediate results have been explicitly named.

The algorithm then generates a typed program p that is decorated with annotation variables, and a set of consistency constraints.

A syntactic soundness theorem guarantees that if substitution $\mathcal{S}$ satisfies the constraints, then the annotated program $\mathcal{S}(p)$ is well-annotated.

We must emphasize that in general, the generated constraint set will have many different solutions. The mechanics of constraint solving, how to choose an (almost) optimal solution, and whether it can be done in a modular way, is an area of immense theoretical as well as practical interest of which our work so far has touched only the surface.

Semantic soundness of our type system is defined with respect to a deterministic, call-by-value abstract machine for annotated programs. The abstract machine and the semantic soundness result are developed in a working companion paper and are not the main subject of this paper.

*Most of the work reported in this paper was carried out while the first author was a postdoctoral fellow at Boston University; he was at Heriot-Watt University at the time when the paper was drafted.

†Partially supported by NSF EIA grant 9806745/9806746/9806747/9806835.

‡Partially supported by Sun grant EDUD-7826-990410-US.

§Partially supported by the DART project (EC grant IST-2001-33477).

¶Partially supported by a Faculty Fellowship of the Wallace E. Carroll School of Management, Boston College.

‖Much of this author's work was completed over the course of a one-semester visit at Harvard University.

TLDI'03, January 18, 2003, New Orleans, Louisiana, USA.

Since it is important to understand the consequences of well-typedness, we briefly summarize the main results.

Annotated programs account for machine-level operations, register moves, stores and loads, procedure calls and stack management. Our annotated language is essentially a typed assembly language [17]. The configurations of the abstract machine contain two kinds of information: one part stores and retrieves values by means of registers and the heap, whereas another part allows the values to be represented directly. A consistent configuration is then one where the two parts coincide. Our formulation of semantic soundness then guarantees that a well-typed consistent configuration is either in a final state, or it evolves into a new well-typed consistent configuration. An important consequence is that live registers are not overwritten, as doing so would destroy consistency.

The performance benefits of flattening are well-known in the ML community [20, 15]. Flattening function-arguments alone can lead to speed ups of 11% percent on average and can reduce memory allocation by 30% on average ([15], p. 70). Our system generalizes argument-flattening by allowing flattening of arbitrary tuples as well as variants. An additional feature of the system is that it provides a type-safe framework for implementing *multiple-value returns* [5]. Finally, our system postpones flattening decisions until the back-end of the compiler when loop nesting-depth and lifetime information can be used in making flattening decisions.

1.1 Contributions

The main contributions of this paper are:

1. We present a type system that supports inter-procedural allocation for virtual registers for ML-like programs with flattening of tuples and variants. Our system supports a *caller-save* calling protocol in which the set of registers to be saved and restored around non-tail-recursive procedure call is customized for the context of the call. No registers are saved before a tail-call.

   The type system presented here has the property that well-typedness ensures that live registers are not overwritten when a program is executed. This is a safety property that is not guaranteed by the notion of type safety developed in [17].

2. We present an algorithm that generates constraints for allocation annotations. We show that the algorithm is sound with respect to the type system. We conjecture that for a well-typed program, the generated constraints will always be solvable.

This paper does not show how to map from virtual registers to physical registers nor does it provide any empirical analysis of heuristics that might be used to support such a mapping. The system presented in this paper does not provide any support for reclaiming heap space.

The system presented here is part of on-going work that was developed as part of a larger effort undertaken in the Church Project[1] to exploit types in the safe and efficient implementation of ML-like languages. We have completed a prototype implementation of the translation and constraint generation phase of the algorithm. We plan to experiment with different methods of dealing with spilling and with various flattening heuristics.

[1] `http://types.bu.edu/`

1.2 Outline of this Paper

The remainder of this paper is organized as follows. In Sect. 2 we present three example programs with annotations that illustrate the properties of our type system. In Sect. 3 we present the type system. In Sect. 4 we present our type inference algorithm and states its soundness. In Sect. 5 we compare this system with others that have appeared in the literature. In Sect. 6 we draw some conclusions and sketch future work.

2. MOTIVATION

In this section we present some example programs to illustrate the properties of the system developed in this paper. We use an ML-like syntax with some syntactic simplifications.

2.1 Inter-procedural Register Allocation and Constructor Flattening

The first example, shown in Fig. 1, illustrates the utility of defunctionalization [18] for typed compilation [23, 8, 10] and it illustrates potential savings to be realized from inter-procedural allocation with flattening. Functions `f` and `g` flow to the same application site but they have different sets of free variables and thus they will have different environment types.

```
let val a = 2
    val b = 3
    fun f(x:int) = x + a * b
    fun g(x:int) = x + b
 in
    (if test then f else g)(343)
end
```

Figure 1: A simple program fragment. Typed compilation can be supported with sum types.

Figures 2 (a) and (b) contain two defunctionalized and annotated versions of the code fragment in figure 1. The annotations on the record type `t1` in the version in Fig. 2 (a) indicate that the fields of a value of this type are stored in the heap and its base address is in register `r1`. The `H` annotation on the sum type `t5` indicates that the injection tag will be stored in the heap. The address of this tag will be stored in `r3`. A well-formedness condition on annotated types will require that the injected value is also stored in the heap (and actually our abstract machine stores it at the location following the tag). Thus, with these annotations, injecting register-resident values of type `t1` or `t3` will entail a store operation. The function `apply` is of type: `t5 X int r4 ->{r5,r6,r7,r8,r9} int r5`. It finds its first argument (an injected environment) in `r3` and its second argument in `r4`. It kills registers `r5` through `r9` and returns its result in `r5`.

Figure 2 (b) shows an alternative annotation of the same program. A value of type `t1` isn't allocated in the heap but instead, its fields are stored in registers `r1` and `r2`. Similarly, the `r3` annotation in type `t3` indicates that the injection tag will be stored in `r3`. Well-formedness conditions impose some restrictions on the injectants but note that they are allowed to be fully inlined in the register file. The function `apply` is of type `t3 X int r4 ->{r5,r6} int r5`.

```
let
  type t1 = {a : int H, b : int H} r1
  type t2 = {a : int H, b : int H} H
  type t3 = {b : int H} r2
  type t4 = {b : int H} H
  type t5 = (t2 +H t4) r3
  fun apply(h : t5, x : int r4) : int r5 =
    case h of
      r : t1 =>
        let val a : int r6 = #a(r)
            val b : int r7 = #b(r)
            val c : int r8 = a * b
            val d : int r5 = x + c
         in d
        end
      r : t3 =>
        let val b : int r9 = #b(r)
            val c : int r5 = x + b
         in c
        end
  val a : int r1 = 3
  val b : int r2 = 4
  val c : t1 = {a = a, b = b}
  val d : t3 = {b = b}
  val e : int r4 = 343
  val h : t5 = if test then
                 inj(1,c)^t5
               else
                 inj(2,d)^t5
 in apply(h,e)
end
```

(a) An annotation with heap-allocated closures.

```
let
  type t1 = {a : int r1, b : int r2} o
  type t2 = {b : int r2} o
  type t3 = (t1 +r3 t2) o
  fun apply(h : t3, x : int r4) : int r5 =
    case h of
      r : t1 =>
        let val a : int r1 = #a(r)   (no-op)
            val b : int r2 = #b(r)   (no-op)
            val c : int r6 = a * b
            val d : int r5 = x + c
         in d
        end
      r : t2 =>
        let val b : int r2 = #b(r)   (no-op)
            val c : int r5 = x + b
         in c
        end
  val a : int r1 = 3
  val b : int r2 = 4
  val c : t1 = {a = a, b = b}        (no-op)
  val d : t2 = {b = b}               (no-op)
  val e : int r4 = 343
  val h : t3 = if test then
                 inj(1,c)^t3
               else
                 inj(2,d)^t3
 in apply(h,e)
end
```

(b) An annotation in which the closures are represented in the register file.

Figure 2: Two annotations of a defunctionalized version of the function in figure 1.

It expects its function-dispatch tag in `r3`. The location(s) of the injected environment depend on the value of the tag. It finds its second argument in `r4`, it kills registers `r5` and `r6` and it returns its result in `r5`.

The expressions that give rise to no run-time operation are marked `(no-op)` on the right. It is worth noting that the operations that are avoided are all relatively expensive off-chip instructions.

A judicious choice of storage annotations can often give rise to a substantial reduction in the number of instructions to be executed. The monomorphic swap function in Fig. 3 is exhibited in two compiled forms in Fig. 4. With the annotation in Fig. 4 (b), the entire swap function is a no-op.

```
let fun swap(x : int, y : bool) = (y,x)
 in
  swap(343,false)
end
```

Figure 3: The Swap function at int × bool**.**

2.2 Non-tail Calls and the Call Stack

The code in figures 2 and 4 contain annotations of tail-recursive function calls. Fig. 5 exhibits an annotation of a non-tail recursive function; factorial. The function has type `({} o X int r1) ->{r2,r3} int r2` which indicates that the function accepts an argument `n` in `r1`, kills registers `r2` and `r3` and returns a result in `r2`. Register `r3` is used as a temporary to hold the value `n - 1`. Since the value of `n` is needed on both sides of the recursive call, register `r1` must be saved on the stack before the call and restored after the function returns. Some care must be taken, however, because in order to make the recursive call, the value in register `r3` must be moved into the argument register `r1` so the order of these operations is important.

3. TYPE SYSTEM

In this section we present the unannotated language and the annotated language.

3.1 The Unannotated Language

The abstract syntax of unannotated types *ut*, function types *uft*, declarations *ud*, expressions *ue* and programs *up* for the unannotated language is presented in Fig. 6 (a). The syntax of expressions has the property that the values of computations are all named. This is a practice of long standing in compiling technology. Recent syntactic specifications of this property have been called *A-normal form* [11], or *named form* [16]. We will refer to it as a *nominal form*. The grammar for expressions provides both tail-recursive function calls: $\text{apply}(x, x)$, and non-tail calls: $\text{let } x = \text{apply}_i(x, x) \text{ in } ue$. Since the program has been defunctionalized, all function definitions have been lifted to the top-level.

The type system is quite standard and the typing rules are omitted.

3.2 The Annotated Language

The abstract syntax of annotations a, annotated types τ,

```
let type t1 = {1 : int H, 2 : bool H} r1
    type t2 = {1 : bool H, 2 : int H} r2
    fun apply(swap : {} o, z : t1) : t2 =
      let val a : int r3 = #1(z)
          val b : bool r4 = #2(z)
          val c : t2 = {1 = b, 2 = a}
       in c
      end
    val a : int r5 = 343
    val b : bool r6 = false
    val c : t1 = {1 = a, 2 = b}
    val swap : {} o = {}
 in
    apply(swap,c)
end
```

(a) Swap with heap-allocated closures. `apply` has type `({} o X t1) ->{r2,r3,r4} t2`.

```
let type t1 = {1 : int r1, 2 : bool r2} o
    type t2 = {1 : bool r2, 2 : int r1} o
    fun apply(swap : {} o, z : t1) : t2 =
      let val a : int r1 = #1(z)        (no-op)
          val b : bool r2 = #2(z)       (no-op)
          val c : t2 = {1 = b, 2 = a}   (no-op)
       in c
      end
     val a : int r1 = 343
     val b : bool r2 = false
     val c : t1 = {1 = a, 2 = b}        (no-op)
     val swap : {} o = {}
 in
    apply(swap,c)
end
```

(b) Swap with flattened records. Swap is a no-op. `apply` has type `({} o X t1) ->{} t2`.

Figure 4: Two annotations of a defunctionalized version of the the swap function (figure 3).

$$
\begin{array}{rcl}
ut & ::= & \text{unit} \mid \text{int} \mid ut \times ut \mid ut + ut \\
uft & ::= & ut \times ut \to ut \\
ud & ::= & c \mid \text{op}(x,x) \mid (x,x) \mid \pi_i(x) \mid \text{inj}(i,x)^{ut} \\
ue & ::= & x \\
 & \mid & \text{apply}_i(x,x) \\
 & \mid & \text{if } x \; ue \; ue \\
 & \mid & \text{case } x \text{ of } x \Rightarrow ue \mid x \Rightarrow ue \\
 & \mid & \text{let } x = ud \text{ in } ue \\
 & \mid & \text{let } x = \text{apply}_i(x,x) \text{ in } ue \\
up & ::= & (\text{apply}_i(x^{ut}, x^{ut}) = ue^{ut})^* \text{ in } ue
\end{array}
$$

(a) A Typed but Unannotated Language.

$$
\begin{array}{rcl}
r \in \text{Reg} & = & \{r_0, r_1, \ldots\} \\
a & ::= & r \mid \mathsf{H} \mid \bullet \\
A & = & \{a, \ldots\} \\
\tau & ::= & t \; a \\
t & ::= & \text{unit} \mid \text{int} \mid \tau \times \tau \mid \tau +^a \tau \\
ft & ::= & \tau \times \tau \xrightarrow{A} \tau \\
z & ::= & x^a \\
d & ::= & c \mid \text{move}(a,z) \mid \text{op}(z,z) \\
 & \mid & (z,z) \mid \pi_i(z) \mid \text{inj}(i,z)^\tau \quad \text{where } \blacktriangleright \; \tau. \\
e & ::= & z \\
 & \mid & \text{apply}_i(z,z) \\
 & \mid & \text{if } z \; e \; e \\
 & \mid & \text{case } x^\tau \text{ of } z \Rightarrow e \mid z \Rightarrow e \\
 & \mid & \text{let } z = d \text{ in } e \\
 & \mid & \text{letcall}_A \; z = \text{apply}_i(\text{move}(a,z), \text{move}(a,z)) \text{ in } e \\
p & ::= & (\text{apply}_i(x^\tau, x^\tau) = e)^* \text{ in } e
\end{array}
$$

(b) A Typed Language with Storage Annotations.

Figure 6: Abstract syntax of types and terms for two languages.

annotated function types ft, declarations d, expressions e and programs p for the annotated language is presented in Fig. 6 (b). The symbol Reg denotes a set of register names. The annotation H indicates that a value is to be stored in the heap. The annotation $\bullet$ indicates that a record or variant is not to be allocated in the heap; instead its components will be in-lined in the register file. When associated with the value *unit*, the $\bullet$ annotation means that the value simply isn't represented at all.

3.2.1 Well-formed Types

The type system depends on two basic well-formedness properties of types. These properties are axiomatized in Fig. 7. We will sometimes write $\text{WFat}(\tau)$ for τ such that $\blacktriangleright \; \tau$ and similarly for $\text{WFht}(\tau)$ and $\triangleright \; \tau$.

The rule $\blacktriangleright \; \tau$ ensures that the root of a value is stored in the register file. If it were not rooted in the register file, it would be inaccessible. Subtrees can be stored in the heap but they must be completely stored in the heap.

Well-formed types obey some properties that are required in the proof of the soundness theorem. Let the predicate $\text{Reg}(a)$ mean that $a = r_i$ for some $i \in I$.

Lemma 3.1. *1. If* $\text{WFat}(t \; a)$ *and* $\text{WFat}(t \; a')$ *then either* $\text{Reg}(a)$ *and* $\text{Reg}(a')$ *or* $a = a' = \bullet$.

2. If $\text{WFat}(t \; r)$ *then* $\text{WFht}(t \; \mathsf{H})$.

□

3.2.2 Annotation of Variable Occurrences

All occurrences of variables include storage annotations. The $\text{move}(a, x^{a'})$ operation moves a value from location a' to location a. In practice, a and a' will usually be registers. It is possible that $a = a'$. In this case, we assume that a *copy-propagation elimination* post-pass will remove these useless instructions.

The most complicated form in the annotated language is the letcall form:

$$\text{letcall}_A \; z = \text{apply}_i \, (\text{move}(a,z), \text{move}(a,z)) \; \text{ in } e$$

This form supports recursive function calls that may not be

```
let
  fun apply(fact : {} o, n : int r1) : int r2 =
    let val a : int r2 = 1
     in if n then a
        else
          let val b : int r3 = n - a
           in letcall{r1} val c:int r2 =
                apply(fact, move(r1,b^r3))
              in
               let val d : int r2 = c * n
                in d
               end
              end
          end
    end
  val e : int r1 = 6
  val fact : {} o = {}
 in
  apply(fact,e)
end
```

Figure 5: A recursive function that isn't tail-recursive.

tail-recursive. It will be explained in the discussion of the typing rules below.

3.3 Typing Rules

The typing rules for our system are defined in Fig. 8.

There are four forms of judgments:

$$E \vdash z : \tau$$
$$E \vdash d : \tau \,!\, A$$
$$E \vdash e : \tau \,!\, A$$
$$\vdash p : \tau \,!\, A$$

The first three forms include a *type environment* E mapping variables to (well-formed) annotated types, and function names to function types. The latter three judgment forms include an *effect set* A. In a derivable judgment, A will conservatively approximate the set of registers written-to in the evaluation of the declaration, expression or program (resp.) component of the judgment.

The Var and Const rules are fairly standard for effect systems. The former has no effect and the latter records a load into register `r`. The Move rule first finds an annotated type $t\ a'$ for z in E. If $t\ a$ is well-formed, then the move is recorded and note that if $a = a'$ then there is no effect.

The $\times$-Intro rule computes the type and effect of allocating a pair. Note that the annotation a'_i is not required to be the same as a_i. When $a = r$ two register-to-heap store instructions must be executed. These store instructions are implicit in this rule. The condition $a'_i \in \{a_i, \mathsf{H}\}$ precludes register to register moves when $a = \bullet$.

Note that a component of a tuple may not have a well-formed type but that an (implicit) coercion can restore well-formedness. For example, if $E(x) = (\text{int } \mathsf{H} \times \text{int } \mathsf{H})\ r_1$ which is well-formed, then $\pi_2(x^{r_1})$ has type int H which is not well-formed. But the expression

$$\text{let } y^{r_2} = \pi_2\,(x^{r_1}) \text{ in } y^{r_2}$$

has type int r_2 and well-formedness is restored.

The Let and Letcall rules make use of an approximation of the liveness properties of the values that arise in the evaluation of their bodies. Our notion of liveness is closely tied to our well-formedness requirement for types. We define $\text{Live} : \tau \to A$, for $\text{WFat}(\tau)$ as follows:

$$\begin{array}{rcl}
\text{Live}(\text{int } r) & = & \{r\} \\
\text{Live}(\text{unit } \bullet) & = & \varnothing \\
\text{Live}((\tau_1 \times \tau_2)\ r) & = & \{r\} \\
\text{Live}((\tau_1 \times \tau_2)\ \bullet) & = & \text{Live}(\tau_1) \cup \text{Live}(\tau_2) \\
\text{Live}((\tau_1 +^{\mathsf{H}} \tau_2)\ r) & = & \{r\} \\
\text{Live}((\tau_1 +^{r} \tau_2)\ \bullet) & = & \{r\} \cup \text{Live}(\tau_1) \cup \text{Live}(\tau_2)
\end{array}$$

Liveness is extended to E and e as follows:

$$\text{Live}\,(E, e) = \{\, a \in \text{Live}(E(x)) \,|\, x \in \text{fvs}(e) \,\}$$

In the Let rule, the registers that may be killed in evaluating the declaration must be disjoint from the live registers of the body.

The subscript A in the Letcall expressions denotes a set of annotations — the *caller save set*. Operationally, the values of the registers in this set will be saved on the stack before the function is applied and they will be restored after the function returns. The save operation will take place *before* the move operations that load the arguments for the function call. The net effect of calling the function is the set of registers killed by the called function, A_0, together with the possible loads of the argument registers $A_1 \cup A_2$. Some of these registers may have been saved (i.e., in A) and will be restored, thus they can be elements of the live variable set of the body. The condition $A \cap \text{Live}(t\ a) = \varnothing$ ensures that register(s) containing part of the result of the function call won't be over-written by the (implicit) restore.

4. INFERRING STORAGE ANNOTATIONS

In this section we present our algorithm for inferring storage annotations.

4.1 Translation and Constraint Generation

The algorithm is presented in Figures 9 and 10. The algorithm accepts a *type scheme environment* $\mathcal{E}$ and a typed but unannotated expression ue and returns a 4-tuple (e, σ, q, C) where e is a typed expression with *annotation variables*, σ is an annotated type scheme, q is a *set expression* and C is a set of *set constraints*. Intuitively, we will look for a solution $\mathcal{S}$ to the set of constraints C. Applying this substitution to ue, σ and q will produce an annotated expression e, an annotated type τ and an effect set A that are guaranteed to be well-typed according to the type system defined in Sect. 3.

Let $\text{AVar} = \{\alpha_1, \alpha_2, \ldots\}$ be a set of *annotation variables*. The set of *annotated type schemes* is the set of annotated types with all annotations a replaced by annotation variables α. Formally, σ and s (corresponding, respectively, to τ and t) are defined as:

$$\begin{array}{rcl}
\sigma & ::= & s\ \alpha \\
s & ::= & \text{unit} \mid \text{int} \mid \sigma \times \sigma \mid \sigma +^{\alpha} \sigma
\end{array}$$

In the remainder of this paper we will redeploy the symbols d, e and p for declarations, expressions and programs (resp.) that are decorated with annotation *variables* rather than real annotations. In the preceding sections, these symbols were reserved for forms with real annotations.

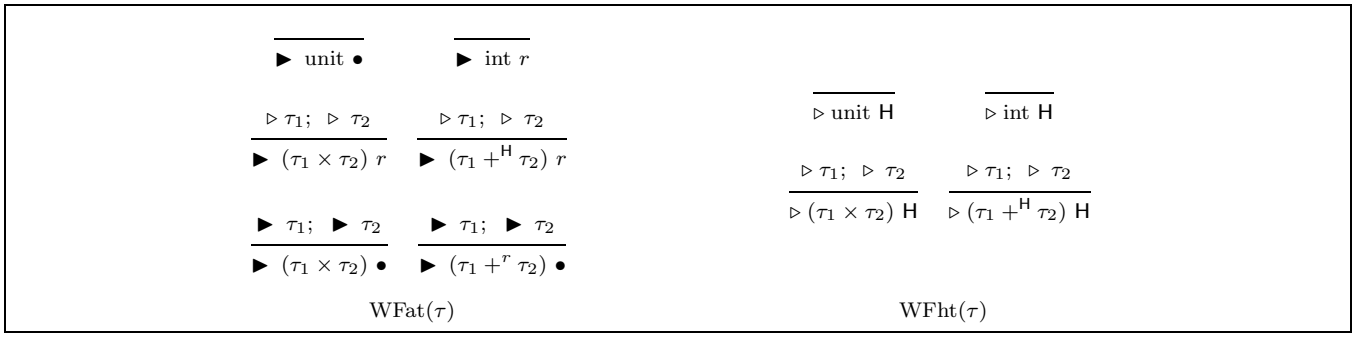

Figure 7: Well-formed Annotated Types and Well-formed Heap Types.

We shall also introduce variables Av that range over *sets* of annotations. It will be convenient to further partition such variables into Avs (for letcall) and Avf (for functions). An *annotated function type scheme* takes the form $\sigma \times \sigma \xrightarrow{\text{Avf}} \sigma$.

The annotation algorithm can generate eleven different types of set constraints. Some of these involve *set expressions* q of the following form:

$$q \quad ::= \quad \text{Avf} \mid \varnothing \mid \{\alpha\} \mid q \cup q \mid q - \text{Avs} \mid q - \{\alpha\}$$

A *set constraint* c is then defined as follows:

$$\begin{array}{rcl} c & ::= & \text{Reg}(\alpha) \mid \alpha_i = \mathsf{H} \mid \alpha_i = \bullet \mid \alpha_i = \alpha_j \\ & \mid & q \subseteq \text{Avf} \mid \alpha \notin q \mid \alpha \notin \text{Avs} \mid \alpha_i \neq \alpha_j \\ & \mid & \alpha \in q \Rightarrow \alpha \in \text{Avs} \mid \alpha \neq \bullet \Rightarrow C \mid \alpha = \bullet \Rightarrow C \end{array}$$

The first five of these are straightforward. The next three involve negation; note that conventional graph color register allocation is expressed in terms of simple inequality constraints. The last three are *conditional constraints*. The first of these, $\alpha \in q \Rightarrow \alpha \in \text{Avs}$ is generated to ensure proper inflation of the caller-save set for a non-tail-recursive function call. The latter two are generated to enforce consistency in the case of flattening.

The annotation algorithm makes use of a number of helper functions that are defined in Fig. 11 of the Appendix. The function Annotate decorates an unannotated term with annotation variables. The function Live computes the set of annotation variables appearing in a type scheme (excluding those that are associated with unit). This function is extended to environments in the usual way:

$$\text{Live}\,(\mathcal{E}, e) = \{\, \alpha \in \text{Live}(\mathcal{E}(x)) \,|\, x \in \text{fvs}(e) \,\}$$

The algorithm will make use of functions UnfoldWFat and UnfoldWFht to generate sets of constraints that are sufficient to ensure that any substitution satisfying the constraints will guarantee the well-formedness of the annotated type obtained from the application of the substitution to the type scheme. The function Decompose accepts two type schemes and generates a set of constraints that ensures equality of the annotated types arising from application of the substitution to the type schemes.

A type scheme environment $\mathcal{E}$ maps variables to annotated type schemes and function names to annotated function type schemes.

The inference algorithm $\mathcal{A}$ has functionality:

$$\begin{array}{rcl} \mathcal{A}(\mathcal{E}, ud) & = & (d, \sigma, q, C) \\ \mathcal{A}(\mathcal{E}, ue) & = & (e, \sigma, q, C) \\ \mathcal{A}(up) & = & (p, \sigma, q, C) \end{array}$$

The algorithm is presented in Figures 9 and 10.

4.2 Substitutions

A substitution $\mathcal{S}$ maps annotation variable α to annotation a and annotation set variables Av to a set of annotations A. Application of substitutions is extended to other constructs in the obvious way; for instance the result of applying a substitution to an annotated type scheme is an annotated type:

$$\begin{array}{rcl} \mathcal{S}(s\ \alpha) & = & \mathcal{S}(s)\ \mathcal{S}(\alpha) \\ \mathcal{S}(\text{unit}) & = & \text{unit} \\ \mathcal{S}(\text{int}) & = & \text{int} \\ \mathcal{S}(\sigma_1 \times \sigma_2) & = & \mathcal{S}(\sigma_1) \times \mathcal{S}(\sigma_2) \\ \mathcal{S}(\sigma_1 +^{\alpha} \sigma_2) & = & \mathcal{S}(\sigma_1) +^{\mathcal{S}(\alpha)} \mathcal{S}(\sigma_2) \\ \mathcal{S}(\sigma_1 \times \sigma_2 \xrightarrow{\text{Avf}} \sigma_3) & = & \mathcal{S}(\sigma_1) \times \mathcal{S}(\sigma_2) \xrightarrow{\mathcal{S}(\text{Avf})} \mathcal{S}(\sigma_3) \end{array}$$

Likewise, the result of applying a substitution to a set expression is a set of annotations:

$$\begin{array}{rcl} \mathcal{S}(\text{Avf}) & = & \mathcal{S}(\text{Avf}) \\ \mathcal{S}(\varnothing) & = & \emptyset \\ \mathcal{S}(\{\alpha\}) & = & \{\mathcal{S}(\alpha)\} \\ \mathcal{S}(q_1 \cup q_2) & = & \mathcal{S}(q_1) \cup \mathcal{S}(q_2) \\ \mathcal{S}(q - \text{Avs}) & = & \mathcal{S}(q) - \mathcal{S}(\text{Avs}) \\ \mathcal{S}(q - \{\alpha\}) & = & \mathcal{S}(q) - \mathcal{S}(\{\alpha\}) \end{array}$$

Note that the set operations on the right-hand side are semantic operations.

We write $\mathcal{S} \models c$ if S satisfies c (defined in the obvious way), and write $\mathcal{S} \models C$ if $\mathcal{S} \models c$ for all $c \in C$.

The following basic properties of substitutions are used in the proof of the soundness theorem.

Lemma 4.1. *1. If* $\text{WFat}(\mathcal{S}(\sigma))$ *then*

$$\text{Live}\,(\mathcal{S}(\sigma)) = \{\, a \in \mathcal{S}(\text{Live}(\sigma)) \,|\, \text{Reg}(a) \,\}.$$

2. If $\mathcal{S} \models \text{Decompose}(\sigma_1, \sigma_2)$ *then* $\mathcal{S}(\sigma_1) = \mathcal{S}(\sigma_2)$.

3. If $\mathcal{S} \models \text{UnfoldWFat}(\sigma)$ *then* $\text{WFat}(\mathcal{S}(\sigma))$.

4. If $\mathcal{S} \models \text{UnfoldWFht}(\sigma)$ *then* $\text{WFht}(\mathcal{S}(\sigma))$.

□

4.3 Correctness

Theorem 4.2 (Soundness). *Let $\mathcal{E}$ be an annotated type scheme environment. Let ud, ue and up be unannotated declarations, expressions and programs (resp.) such that* $\mathrm{fvs}(ud) \subseteq \mathrm{Dom}(\mathcal{E})$ *and* $\mathrm{fvs}(ue) \subseteq \mathrm{Dom}(\mathcal{E})$. *Let $\mathcal{S}$ be a substitution.*

1. *If* $\mathcal{A}(\mathcal{E}, ud) = (d, \sigma, q, C)$, $\mathcal{S} \models C$ *and* $\mathrm{WFat}(\mathcal{S}(\mathcal{E}))$ *then*
$$\mathcal{S}(\mathcal{E}) \vdash \mathcal{S}(d) : \mathcal{S}(\sigma)\ !\ \mathcal{S}(q).$$
2. *If* $\mathcal{A}(\mathcal{E}, ue) = (e, \sigma, q, C)$, $\mathcal{S} \models C$ *and* $\mathrm{WFat}(\mathcal{S}(\mathcal{E}))$ *then*
$$\mathcal{S}(\mathcal{E}) \vdash \mathcal{S}(e) : \mathcal{S}(\sigma)\ !\ \mathcal{S}(q).$$
3. *If* $\mathcal{A}(up) = (p, \sigma, q, C)$ *where* $\mathcal{S} \models C$ *then*
$$\vdash \mathcal{S}(p) : \mathcal{S}(\sigma)\ !\ \mathcal{S}(q).$$

□

Proof. The proof, which for (2) uses induction in ue, is by a case analysis in the definition of $\mathcal{A}$. None of the cases presents any conceptual difficulties; Lemma 4.1 is heavily used. □

4.4 Constraint Satisfaction

We conjecture that if $\mathcal{A}(up) = (p, \sigma, q, C)$ then it will always be possible to find an $\mathcal{S}$ such that $\mathcal{S} \models C$. Essentially, this can be done (albeit sub-optimally) by *(i)* not doing any flattening; and *(ii)* whenever possible, mapping the α to different registers. We have a proof of this claim for a slightly altered system but we have not pushed the proof through for the current system. Actually, what could cause trouble is

- that we don't put a move instruction around a function call;
- that we don't allow the possibility of moving the arguments to a tail-recursive function call.

5. RELATED WORK

The literature on efficient representations of functions in higher-order programming languages is extensive [12, 4, 3, 19, 22, 21, 9].

Our work is most closely related to, and was partially inspired by, the work of Johan Agat [2, 1]. Agat develops a typed λ-calculus, λ_R, which is an effect-based system that tracks liveness properties and supports inter-procedural allocation of virtual registers. The λ_R system can also track allocation of records in the register file. It provides more flexibility in allowing higher-order functions to accept function arguments that expect arguments in different registers (what Agat calls register polymorphism). This is achieved by the insertion of *wrapper* functions. The move instruction in our system implements a kind of shallow subtyping that forces argument functions to expect their arguments in the same registers.

Although the system presented here has the same general goal as Agat's (type-safe, inter-procedural register allocation), there are important differences. We consider the problem in the context of a particular position in the back-end of a type-directed compiler — immediately after typed defunctionalization. In this setting all higher-order functions have been eliminated so we are able to formalize the system as a first-order system augmented with products and sums. As a consequence, we believe that our system is somewhat simpler. For example, our semantic soundness result is proved with respect to a single abstract machine semantics. The soundness result presented in [1] involves three semantics: a standard semantics (Std-semantics), a store semantics (SR-semantics) and an agreement semantics (A-semantics).

The work presented here also differs from Agat's in that it develops an algorithm for inserting annotations and move instructions. The type system of λ_R can determine when load and move instructions have been safely inserted, but it did not attempt to show how or where to insert them.

Our work also has close parallels with typed assembly language [17]. We do not present a complete compiler and prove its type safety. Instead we present only a type system and the annotation phase of a compiler. Well-typedness in our type system guarantees that live registers will not be over-written. Although the compiler presented in [17] generates safe code that does not over-write live registers, the type system of TAL allows one to over-write a live register as long as the new value is of the same type as the old one.

Boquist [7] developed an inter-procedural register allocation algorithm for a lazy functional language. His algorithm introduced a technique to optimize the placement of register save and restore instructions and extended Chaitin-style graph coloring with inter-procedural coalescing and a restricted form of live range splitting.

6. CONCLUSION

We have presented an annotated type system that supports inter-procedural allocation of virtual registers and supports flattening of tuples and variants. We have also presented an algorithm for inserting annotations and whatever move instructions are required to preserve type safety. This work contributes to our over-arching goal of producing an efficient compiler for an ML-like language that is certifiably type-safe.

Since our framework relegates the essential allocation decisions to the constraint solver, the most urgent task for future work is to implement such, and conduct experiments so as to discover heuristics for producing quality code. In particular, a crucial choice is when to flatten a representation (i.e., assigning $\bullet$s). Empirical evidence suggests that arguments to functions should be flattened. However, if there are many arguments, then the called function will exert a good deal of register pressure. Loop-nesting depth information may be a good guide. Also, even though constraint *generation* is modular, it remains to be seen whether also constraint *solving* can be done in a modular way, or whether we would need the whole program to be present.

There are many other important issues that remain to be addressed. Allowing a callee-save protocol would be an improvement. The problem of mapping from virtual registers to physical registers will require more work. Spilling is problematic because the analysis on the pre-spilled term will not be sound for the transformed program that contains the spill code. It is possible that the effects of the spill code can be restricted by using the letcall construct to save spilled registers on the stack.

Even though it is common for compilers of higher-order languages to defunctionalize, one could ask whether the techniques can be used directly in a higher-order setting. Actu-

ally, we aimed for that originally, but met substantial difficulties we couldn't see how to overcome.

Acknowledgments

The authors gratefully acknowledge the contributions made by Brendan Connell in implementing a prototype of the translation and constraint generation phase of the annotation algorithm. Our formulation of the system on already defunctionalized code was inspired by a Church Project seminar given by Anindya Banerjee [6].

APPENDIX

Figure 11 contains definitions for several of the auxiliary functions used in the paper.

A. REFERENCES

[1] J. Agat. *A Typed Functional Language for Expressing Register Usage*. Ph.D. thesis, Chalmers University of Technology and Goteborg University, Sept. 1998.

[2] J. Agat. Types for register allocation. In *Proc. of IFL'97: 9th Int'l Workshop Implementation Functional Languages*, vol. 1467 of *LNCS*, pp. 92–111, St. Andrews, Scotland, Sept. 1998. Springer-Verlag.

[3] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Conf. Rec. 16th Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 293–302, 1989.

[5] J. M. Ashley and R. K. Dybvig. An efficient implementation of multiple return values in scheme. In *LISP and Functional Programming*, pp. 140–149, 1994.

[6] A. Banerjee, N. Heintze, and J. Riecke. Design and correctness of program transformations based on control-flow analysis. In *Theoretical Aspects Comput. Softw. : Int'l Conf.*, vol. 2215 of *LNCS*, pp. 420–447, Berlin, Oct. 2001. Springer-Verlag.

[7] U. Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. Ph.D. thesis, Chalmers University of Technology and Goteborg University, Apr. 1999.

[8] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*, pp. 56–71. Springer-Verlag, 2000.

[9] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *Proc. 1997 Int'l Conf. Functional Programming*, pp. 11–24. ACM Press, 1997.

[10] A. Dimock, I. Westmacott, R. Muller, F. Turbak, and J. B. Wells. Functioning without closure: Type-safe customized function representations for Standard ML. In *Proc. 6th Int'l Conf. Functional Programming*, pp. 14–25. ACM Press, 2001.

[11] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. ACM SIGPLAN '93 Conf. Prog. Lang. Design & Impl.*, 1993.

[12] D. Kranz, R. Kelsey, J. A. Rees, P. Hudak, J. Philbin, and N. I. Adams. Orbit: An optimizing compiler for Scheme. In *Proc. SIGPLAN '86 Symp. Compiler Construction*, pp. 219–233, 1986.

[13] J. M. Lucassen. *Types and Effects, towards the Integration of Functional and Imperative Programming*. Ph.D. thesis, MIT Laboratory for Computer Science, 1987.

[14] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conf. Rec. 15th Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 47–57, 1988.

[15] G. Morrisett. *Compiling with Types*. Ph.D. thesis, Carnegie Mellon University, 1995.

[16] G. Morrisett and R. Harper. Semantics of memory management for polymorphic languages. Technical Report CMU-CS-96-176, Carnegie Mellon University, Sept. 1996.

[17] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Prog. Langs. & Systs.*, 21(3):528–569, May 1999.

[18] J. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pp. 717–740, 1972.

[19] Z. Shao and A. Appel. Space-efficient closure representations. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, 1994.

[20] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 116–129, La Jolla, CA, Jun 1995.

[21] P. Steckler and M. Wand. Lightweight closure conversion. *ACM Trans. on Prog. Langs. & Systs.*, 19(1):48–86, Jan. 1997.

[22] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. Prog. Lang. Design & Impl.*, 1996.

[23] A. P. Tolmach and D. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *J. Funct. Programming*, 8(4):367–412, 1998.

$E \vdash z : \tau$

$$\frac{E(x) = t\ a;\ \ \blacktriangleright\ t\ a}{E \vdash x^a : t\ a}\ \text{Var}$$

$E \vdash d : \tau\ !\ A$

$$\frac{}{E \vdash c : \text{int}\ r\ !\ \{r\}}\ \text{Const}$$

$$\frac{E \vdash z : t\ a';\ \ \blacktriangleright\ t\ a}{E \vdash \text{move}(a, z) : t\ a\ !\ \{a\} - \{a'\}}\ \text{Move} \qquad \frac{E \vdash z_1 : \text{int}\ r_1;\ E \vdash z_2 : \text{int}\ r_2}{E \vdash \text{op}(z_1, z_2) : \text{int}\ r\ !\ \{r\}}\ \text{Primop}$$

$$\frac{E \vdash z_1 : t_1\ a_1;\ E \vdash z_2 : t_2\ a_2;\ \ \blacktriangleright\ (t_1\ a_1' \times t_2\ a_2')\ a;\ a_i' \in \{a_i, \mathsf{H}\}}{E \vdash (z_1, z_2) : (t_1\ a_1' \times t_2\ a_2')\ a\ !\ \{a\}}\ \times\text{-Intro}$$

$$\frac{E \vdash z : (t_1\ a_1 \times t_2\ a_2)\ a;\ \ \blacktriangleright\ t_i\ a_i';\ a_i \in \{a_i', \mathsf{H}\}}{E \vdash \pi_i(z) : t_i\ a_i'\ !\ \{a_i'\} - \{a_i\}}\ \times\text{-Elim}$$

$$\frac{\tau = (t_1\ a_1' +^{a'} t_2\ a_2)\ a \quad E \vdash z : t_1\ a_1;\ \ \blacktriangleright\ \tau;\ a_1' \in \{a_1, \mathsf{H}\}}{E \vdash \text{inj}(1, z)^\tau : \tau\ !\ \{a, a'\}}\ +\text{-Intro}_1 \qquad \frac{\tau = (t_1\ a_1 +^{a'} t_2\ a_2')\ a \quad E \vdash z : t_2\ a_2;\ \ \blacktriangleright\ \tau;\ a_2' \in \{a_2, \mathsf{H}\}}{E \vdash \text{inj}(2, z)^\tau : \tau\ !\ \{a, a'\}}\ +\text{-Intro}_2$$

$E \vdash e : \tau\ !\ A$

$$\frac{E \vdash z : \tau}{E \vdash z : \tau\ !\ \varnothing}\ \text{Var}$$

$$\frac{E(\text{apply}_i) = \tau_1 \times \tau_2 \xrightarrow{A} \tau \quad E \vdash z_1 : \tau_1;\ E \vdash z_2 : \tau_2}{E \vdash \text{apply}_i(z_1, z_2) : \tau\ !\ A}\ \text{Tail Call} \qquad \frac{E \vdash z : \text{int}\ r;\ E \vdash e_1 : \tau\ !\ A_1;\ E \vdash e_2 : \tau\ !\ A_2}{E \vdash \text{if}\ z\ e_1\ e_2 : \tau\ !\ A_1 \cup A_2}\ \text{If}$$

$$\frac{\begin{array}{c}E(x) = \tau = (t_1\ a_1 +^{a'} t_2\ a_2)\ a \\ A = A_1 \cup A_2 \cup (\{a_1'\} - \{a_1\}) \cup (\{a_2'\} - \{a_2\}) \\ \text{for } i \in \{1, 2\} : E[x_i : t_i\ a_i'] \vdash e_i : \tau'\ !\ A_i;\ \ \blacktriangleright\ t_i\ a_i';\ a_i \in \{a_i', \mathsf{H}\}\end{array}}{E \vdash \text{case}\ x^\tau\ \text{of}\ x_1^{a_1'} \Rightarrow e_1 \mid x_2^{a_2'} \Rightarrow e_2 : \tau'\ !\ A}\ \text{Case}$$

$$\frac{E \vdash d : t\ a\ !\ A_0;\ E[x : t\ a] \vdash e : \tau\ !\ A;\ \text{Live}(E, e) \cap A_0 = \varnothing}{E \vdash \text{let}\ x^a = d\ \text{in}\ e : \tau\ !\ A_0 \cup A}\ \text{Let}$$

$$\frac{\begin{array}{c}E(\text{apply}_i) = (t_1\ a_1) \times (t_2\ a_2) \xrightarrow{A_0} t\ a \\ a_1 \notin (\text{Live}(t_2\ a_2') \cup \{a_2\});\ a_2 \notin (\text{Live}(t_1\ a_1') \cup \{a_1\}) \\ E \vdash \text{move}(a_1, x_1^{a_1'}) : (t_1\ a_1)\ !\ A_1;\ E \vdash \text{move}(a_2, x_2^{a_2'}) : (t_2\ a_2)\ !\ A_2 \\ E[x : t\ a] \vdash e : \tau\ !\ A' \\ A_0' = (A_0 \cup A_1 \cup A_2) - A;\ A_0' \cap \text{Live}(E, e) = \varnothing;\ A \cap \text{Live}(t\ a) = \varnothing\end{array}}{E \vdash \text{letcall}_A\ x^a = \text{apply}_i(\text{move}(a_1, x_1^{a_1'}), \text{move}(a_2, x_2^{a_2'}))\ \text{in}\ e : \tau\ !\ A_0' \cup A'}\ \text{Letcall}$$

$\vdash p : \tau\ !\ A$

$$\frac{\begin{array}{c}E(\text{apply}_i) = \tau_i' = \tau_{1i} \times \tau_{2i} \xrightarrow{A_i} \tau_i;\ \text{WFat}(\tau_{1i});\ \text{WFat}(\tau_{2i});\ \text{WFat}(\tau_i') \\ E[x_{1i} : \tau_{1i}, x_{2i} : \tau_{2i}] \vdash e_i : \tau_i\ !\ A_i';\ \{\, a \in A_i' \mid \text{Reg}(a) \,\} \subseteq A_i \\ E \vdash e : \tau\ !\ A\end{array}}{\vdash (\text{apply}_i(x_{1i}^{\tau_{1i}}, x_{2i}^{\tau_{2i}}) = e_i)^*\ \text{in}\ e : \tau\ !\ A}\ \text{Program}$$

Figure 8: Typing rules for the annotated language.

$\mathcal{A}(\mathcal{E}, c) = (c, \text{int } \alpha, \{\alpha\}, \{\text{Reg}(\alpha)\})$ where α is fresh

$\mathcal{A}(\mathcal{E}, \text{op}(x_1, x_2)) =$
  let $\mathcal{E}(x_1) = \text{int } \alpha_1$
    $\mathcal{E}(x_2) = \text{int } \alpha_2$
    α be fresh
  in
    $(\text{op}(x_1^{\alpha_1}, x_2^{\alpha_2}), \text{int } \alpha, \{\alpha\}, \{\text{Reg}(\alpha)\})$

$\mathcal{A}(\mathcal{E}, (x_1, x_2)) =$
  let $\mathcal{E}(x_1) = s_1\ \alpha_1$
    $\mathcal{E}(x_2) = s_2\ \alpha_2$
    $\alpha, \alpha_1', \alpha_2'$ be fresh
    $C = \{\ \alpha = \bullet \Rightarrow \{\alpha_1 = \alpha_1', \alpha_2 = \alpha_2'\}, \alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha), \alpha_1' = \mathsf{H}, \alpha_2' = \mathsf{H}\}\} \cup$
      (if $s_1 \neq$ unit then $\{\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha_1)\}\}$ else $\varnothing$) $\cup$
      (if $s_2 \neq$ unit then $\{\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha_2)\}\}$ else $\varnothing$)
  in
    $((x_1^{\alpha_1}, x_2^{\alpha_2}), (s_1\ \alpha_1' \times s_2\ \alpha_2')\ \alpha, \{\alpha\}, C)$

$\mathcal{A}(\mathcal{E}, \pi_i(x)) =$
  let $\mathcal{E}(x) = (s_1\ \alpha_1 \times s_2\ \alpha_2)\ \alpha$
    α_i' be fresh
    $C = \{\alpha = \bullet \Rightarrow \{\alpha_i' = \alpha_i\}\} \cup$
      (if $s_i =$ unit then $\{\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha), \alpha_i' = \bullet\}\}$ else $\{\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha), \text{Reg}(\alpha_i')\}\}$)
  in
    $(\pi_i(x^\alpha), s_i\ \alpha_i', \{\alpha_i'\} - \{\alpha_i\}, C)$

$\mathcal{A}(\mathcal{E}, \text{inj}(1, x)^{(ut_1 + ut_2)}) =$
  let $\mathcal{E}(x) = s_1\ \alpha_1$
    $s_2\ \alpha_2 = \text{Annotate}(ut_2)$
    $\alpha, \alpha', \alpha_1'$ be fresh
    $\sigma = (s_1\ \alpha_1' +^{\alpha'} s_2\ \alpha_2)\ \alpha$
    $C = \{\ \alpha = \bullet \Rightarrow \{\alpha_1 = \alpha_1', \text{Reg}(\alpha')\} \cup \text{UnfoldWFat}(s_2\ \alpha_2),$
      $\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha), \alpha_1' = \mathsf{H}, \alpha' = \mathsf{H}\} \cup \text{UnfoldWFht}(s_2\ \alpha_2)\} \cup$
      (if $s_1 \neq$ unit then $\{\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha_1)\}\}$ else $\varnothing$)
  in
    $(\text{inj}(1, x^{\alpha_1})^\sigma, \sigma, \{\alpha\} \cup \{\alpha'\}, C)$

$\mathcal{A}(\mathcal{E}, x) =$
  let $\mathcal{E}(x) = s\ \alpha$
    α' be fresh
  in
    $(\text{let } x^{\alpha'} = \text{move}(\alpha', x^\alpha) \text{ in } x^{\alpha'}, s\ \alpha', \{\alpha'\} - \{\alpha\}, \text{UnfoldWFat}(s\ \alpha'))$

$\mathcal{A}(\mathcal{E}, \text{apply}_i(x_1, x_2)) =$
  let $\mathcal{E}(\text{apply}_i) = \sigma_1 \times \sigma_2 \xrightarrow{\text{Avf}} \sigma_3$
    $\mathcal{E}(x_1) = \sigma_1' = s_1\ \alpha_1$
    $\mathcal{E}(x_2) = \sigma_2' = s_2\ \alpha_2$
    α_3' be fresh
    $C = \text{Decompose}(\sigma_1', \sigma_1) \cup \text{Decompose}(\sigma_2', \sigma_2)$
  in
    $(\text{apply}_i(x_1^{\alpha_1}, x_2^{\alpha_2}), \sigma_3, \text{Avf}, C)$

$\mathcal{A}(\mathcal{E}, \text{if } x\ ue_1\ ue_2) =$
  let $\mathcal{E}(x) = \text{int } \alpha$
    $(e_1, \sigma_1, q_1, C_1) = \mathcal{A}(\mathcal{E}, ue_1)$
    $(e_2, \sigma_2, q_2, C_2) = \mathcal{A}(\mathcal{E}, ue_2)$
    $C = C_1 \cup C_2 \cup \text{Decompose}(\sigma_1, \sigma_2) \cup \{\text{Reg}(\alpha)\}$
  in
    $(\text{if } x^\alpha\ e_1\ e_2, \sigma_1, q_1 \cup q_2, C)$

Figure 9: Annotation Algorithm (Part 1).

$\mathcal{A}(\mathcal{E}, \text{case } x \text{ of } x_1 \Rightarrow ue_1 \mid x_2 \Rightarrow ue_2) =$
  let $\mathcal{E}(x) = \sigma = (s_1\ \alpha_1 +^{\alpha'}\ s_2\ \alpha_2)\ \alpha$
    α'_1, α'_2 be fresh
    $\mathcal{E}_i = \mathcal{E}[x_i : s_i\ \alpha'_i]$
    $(e_i, \sigma_i, q_i, C_i) = \mathcal{A}(\mathcal{E}_i, ue_i)$
    $C = C_1 \cup C_2 \cup \text{Decompose}(\sigma_1, \sigma_2) \cup$
      $\{\alpha = \bullet \Rightarrow \{\alpha_1 = \alpha'_1, \alpha_2 = \alpha'_2\}\} \cup$
      $(\text{if } s_1 \neq \text{unit then } \{\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha'_1)\}\} \text{ else } \{\alpha \neq \bullet \Rightarrow \{\alpha'_1 = \bullet\}\}) \cup$
      $(\text{if } s_2 \neq \text{unit then } \{\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha'_2)\}\} \text{ else } \{\alpha \neq \bullet \Rightarrow \{\alpha'_2 = \bullet\}\})$
  in
    $(\text{case } x^\sigma \text{ of } x_1^{\alpha'_1} \Rightarrow e_1 \mid x_2^{\alpha'_2} \Rightarrow e_2, \sigma_1, q_1 \cup q_2 \cup (\{\alpha'_1\} - \{\alpha_1\}) \cup (\{\alpha'_2\} - \{\alpha_2\}), C)$

$\mathcal{A}(\mathcal{E}, \text{let } x = ud \text{ in } ue) =$
  let $(d, s_1\ \alpha_1, q_1, C_1) = \mathcal{A}(\mathcal{E}, ud)$
    $(e, \sigma, q_2, C_2) = \mathcal{A}(\mathcal{E}[x : s_1\ \alpha_1], ue)$
    $C = \{\, \alpha \notin q_1 \,|\, \alpha \in \text{Live}(\mathcal{E}, e) \,\} \cup C_1 \cup C_2$
  in
    $(\text{let } x^{\alpha_1} = d \text{ in } e, \sigma, q_1 \cup q_2, C)$

$\mathcal{A}(\mathcal{E}, \text{let } x_1 = \text{apply}_i(x_2, x_3) \text{ in } ue) =$
  let $\mathcal{E}(\text{apply}_i) = (s_2\ \alpha_2) \times (s_3\ \alpha_3) \xrightarrow{\text{Avf}} (s_1\ \alpha_1)$
    $\mathcal{E}(x_2) = (s'_2\ \alpha'_2)$
    $\mathcal{E}(x_3) = (s'_3\ \alpha'_3)$
    Avs be fresh
    $q_1 = \text{Avf} \cup (\{\alpha_2\} - \{\alpha'_2\}) \cup (\{\alpha_3\} - \{\alpha'_3\})$
    $(e, \sigma, q', C') = \mathcal{A}(\mathcal{E}[x : s_1\ \alpha_1], ue)$
    $C = \text{Decompose}(s_2\ \alpha_2, s'_2\ \alpha_2) \cup \text{Decompose}(s_3\ \alpha_3, s'_3\ \alpha_3) \cup$
      $C' \cup \{\, \alpha \notin \text{Avs} \,|\, \alpha \in \text{Live}(s_1\ \alpha_1) \,\} \cup \{\, \alpha \in q_1 \Rightarrow \alpha \in \text{Avs} \,|\, \alpha \in \text{Live}(\mathcal{E}, e) \,\} \cup$
      $\{\, \alpha_2 \neq \alpha \,|\, \alpha \in \text{Live}(s_3\ \alpha'_3) \cup \{\alpha_3\} \,\} \cup \{\, \alpha_3 \neq \alpha \,|\, \alpha \in \text{Live}(s_2\ \alpha'_2) \cup \{\alpha_2\} \,\}$
  in
    $(\text{letcall}_{\text{Avs}}\ x^{\alpha_1} = \text{apply}_i(\text{move}(\alpha_2, x_2^{\alpha'_2}), \text{move}(\alpha_3, x_3^{\alpha'_3})) \text{ in } e, \sigma, (q_1 - \text{Avs}) \cup q', C)$

$\mathcal{A}(\{\, \text{apply}_i(x_{1i}^{ut_{1i}}, x_{2i}^{ut_{2i}}) = ue_i^{ut_i} \,|\, i \in I \,\} \text{ in } ue) =$
  let $\sigma_{1i} = \text{Annotate}(ut_{1i})$
    $\sigma_{2i} = \text{Annotate}(ut_{2i})$
    $\sigma_i = \text{Annotate}(ut_i)$
    Avf_i be fresh
    $\mathcal{E} = \{\, \text{apply}_i : \sigma_{1i} \times \sigma_{2i} \xrightarrow{\text{Avf}_i} \sigma_i \,|\, i \in I \,\}$
    $(e_i, \sigma'_i, q_i, C_i) = \mathcal{A}(\mathcal{E}[x_{1i} : \sigma_{1i}, x_{2i} : \sigma_{2i}], ue_i)$
    $(e, \sigma, q, C') = \mathcal{A}(\mathcal{E}, ue)$
    $C = \bigcup C_i \cup \text{UnfoldWFat}(\sigma_{1i}) \cup \text{UnfoldWFat}(\sigma_{2i}) \cup \text{UnfoldWFat}(\sigma_i) \cup \text{Decompose}(\sigma'_i, \sigma_i) \cup \{q_i \subseteq \text{Avf}_i\} \cup C'$
  in
    $(\{\, \text{apply}_i(x_{1i}^{\sigma_{1i}}, x_{2i}^{\sigma_{2i}}) = e_i \,|\, i \in I \,\} \text{ in } e, \sigma, q, C)$

Figure 10: Annotation Algorithm (Part 2).

Annotate : $ut \to \sigma$

$$
\begin{array}{rcll}
\text{Annotate}(\text{unit}) & = & \text{unit } \alpha & \alpha \text{ fresh} \\
\text{Annotate}(\text{int}) & = & \text{int } \alpha & \alpha \text{ fresh} \\
\text{Annotate}(ue_1 \times ue_2) & = & (\text{Annotate}(ue_1) \times \text{Annotate}(ue_2))\ \alpha & \alpha \text{ fresh} \\
\text{Annotate}(ue_1 + ue_2) & = & (\text{Annotate}(ue_1) +^{\alpha_1} \text{Annotate}(ue_2))\ \alpha_2 & \alpha_1, \alpha_2 \text{ fresh}
\end{array}
$$

Live : $\sigma \to A$

$$
\begin{array}{rcl}
\text{Live}(\text{int } \alpha) & = & \{\alpha\} \\
\text{Live}(\text{unit } \alpha) & = & \varnothing \\
\text{Live}((\sigma_1 \times \sigma_2)\ \alpha) & = & \{\alpha\} \cup \text{Live}(\sigma_1) \cup \text{Live}(\sigma_2) \\
\text{Live}((\sigma_1 +^{\alpha'} \sigma_2)\ \alpha) & = & \{\alpha, \alpha'\} \cup \text{Live}(\sigma_1) \cup \text{Live}(\sigma_2)
\end{array}
$$

UnfoldWFat : $\sigma \to C$

$$
\begin{array}{rcl}
\text{UnfoldWFat}(\text{int } \alpha) & = & \{\text{Reg}(\alpha)\} \\
\text{UnfoldWFat}(\text{unit } \alpha) & = & \{\alpha = \bullet\} \\
\text{UnfoldWFat}((\sigma_1 \times \sigma_2)\ \alpha) & = & \{\alpha = \bullet \Rightarrow \text{UnfoldWFat}(\sigma_1) \cup \text{UnfoldWFat}(\sigma_2)\} \cup \\
 & & \{\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha)\} \cup \text{UnfoldWFht}(\sigma_1) \cup \text{UnfoldWFht}(\sigma_2)\} \\
\text{UnfoldWFat}((\sigma_1 +^{\alpha'} \sigma_2)\ \alpha) & = & \{\alpha = \bullet \Rightarrow \{\text{Reg}(\alpha')\} \cup \text{UnfoldWFat}(\sigma_1) \cup \text{UnfoldWFat}(\sigma_2)\} \cup \\
 & & \{\alpha \neq \bullet \Rightarrow \{\text{Reg}(\alpha), \alpha' = \mathsf{H}\} \cup \text{UnfoldWFht}(\sigma_1) \cup \text{UnfoldWFht}(\sigma_2)\}
\end{array}
$$

UnfoldWFht : $\sigma \to C$

$$
\begin{array}{rcl}
\text{UnfoldWFht}(\text{int } \alpha) & = & \{\alpha = \mathsf{H}\} \\
\text{UnfoldWFht}(\text{unit } \alpha) & = & \{\alpha = \mathsf{H}\} \\
\text{UnfoldWFht}((\sigma_1 \times \sigma_2)\ \alpha) & = & \{\alpha = \mathsf{H}\} \cup \text{UnfoldWFht}(\sigma_1) \cup \text{UnfoldWFht}(\sigma_2) \\
\text{UnfoldWFht}((\sigma_1 +^{\alpha'} \sigma_2)\ \alpha) & = & \{\alpha = \mathsf{H}, \alpha' = \mathsf{H}\} \cup \text{UnfoldWFht}(\sigma_1) \cup \text{UnfoldWFht}(\sigma_2)
\end{array}
$$

Decompose : $\sigma \times \sigma \to C$

$$
\begin{array}{rcl}
\text{Decompose}(\text{int } \alpha, \text{int } \alpha') & = & \{\alpha = \alpha'\} \\
\text{Decompose}(\text{unit } \alpha, \text{unit } \alpha') & = & \{\alpha = \alpha'\} \\
\text{Decompose}((\sigma_1 \times \sigma_2)\ \alpha, (\sigma_1' \times \sigma_2')\ \alpha') & = & \{\alpha = \alpha'\} \cup \text{Decompose}(\sigma_1, \sigma_1') \cup \text{Decompose}(\sigma_2, \sigma_2') \\
\text{Decompose}((\sigma_1 +^{\alpha_0} \sigma_2)\ \alpha, (\sigma_1' +^{\alpha_0'} \sigma_2')\ \alpha') & = & \{\alpha = \alpha', \alpha_0 = \alpha_0'\} \cup \text{Decompose}(\sigma_1, \sigma_1') \cup \text{Decompose}(\sigma_2, \sigma_2')
\end{array}
$$

Figure 11: Auxilliary Functions.