Higher-Order Symb Comput (2008) 21: 333–358
DOI 10.1007/s10990-008-9035-3

Flattening tuples in an SSA intermediate representation

Lukasz Ziarek · Stephen Weeks · Suresh Jagannathan

Published online: 30 September 2008

Abstract For functional programs, unboxing aggregate data structures such as tuples removes memory indirections and frees dead components of the decoupled structures. To explore the consequences of such optimizations in a whole-program compiler, this paper presents a tuple flattening transformation and a framework that allows the formal study and comparison of different flattening schemes.

We present our transformation over functional SSA, a simply-typed, monomorphic language and show that the transformation is type-safe. The flattening algorithm defined by our transformation has been incorporated into MLton, a whole-program, optimizing compiler for SML. Experimental results indicate that aggressive tuple flattening can lead to substantial improvements in runtime performance, a reduction in code size, and a decrease in total allocation without a significant increase in compilation time.

Keywords Compilation · Optimization · Unboxing · Flattening · Tuples · SSA

1 Introduction

Unboxing is a method exploited by optimizing compilers to remove memory indirections and unnecessary data. Unboxing optimizations improve run-time performance while preserving behavior as well as type safety of programs. This paper presents a new SSA transformation for flattening tuples in a whole-program compilation environment. Our transformation differs from classic unboxing methods and deforestation algorithms, utilizing whole-program analysis to perform unboxing on a simple first-order intermediate language. To our

L. Ziarek (✉) · S. Jagannathan
Department of Computer Science, Purdue University, West Lafayette, USA
e-mail: lziarek@cs.purdue.edu

S. Jagannathan
e-mail: suresh@cs.purdue.edu

S. Weeks
Jane Street Capital, New York, USA
e-mail: sweeks@sweeks.com

knowledge, our transformation is the first to study the benefits of whole-program compilation for unboxing.

The core of the flattening transformation revolves around passing both boxed and unboxed representations where needed. An unboxed representation is created by prefetching tuple elements, i.e., selecting out the tuple's structure. Where feasible, flattening a tuple releases the association between the tuple's constituent components, potentially enabling improved memory utilization through the elimination of useless components and corresponding selects. Our optimization strategy relies on a labeling of tuples, indicating which tuples should be flattened. The correctness of a transformation induced by a labeling may require that multiple representations of a tuple be preserved; to ensure this, both boxed and unboxed representations are passed to all tuple use-sites.[1] The degree to which flattening alleviates memory indirections determines the efficiency of the optimization.

To motivate our approach, consider the following two versions of the functions 'CoordinateAdd':

```
fun CoordinateAdd(f, pair1, pair2) =
  let val x1 = # 1 pair1
      val x2 = # 1 pair2
  in f(x1 + x2, pair1)
  end

fun CoordinateAddT(f, p1x, p1y, p2x, p2y) =
  let val x1 = p1x
      val x2 = p2x
  in f(x1 + x2, p1x, p1y)
  end
```

Suppose $\texttt{pair}_1$ and $\texttt{pair}_2$ are pairs in a typical (x, y) coordinate system and all possible candidates for f are known. Unboxing both pairs, thereby passing the function CoordinateAdd four arguments and the function f, eliminates the need for select statements of the form, $x_1 = \#1\ \texttt{pair}_1$. If it can be determined that the function f's arguments can also be flattened, this transformation permits the compiler to pass each of the four arguments, representing the decoupled elements of $\texttt{pair}_1$ and $\texttt{pair}_2$, in registers instead of providing two heap allocated objects, as seen in $\texttt{CoordinateAdd}_\texttt{T}$. When the pairs are flattened, the association between x and y dissolves, allowing the y coordinate of $\texttt{pair}_2$ to be dropped through useless variable elimination [17, 26]. Breaking the correlation between x and y coordinates within the local context of CoordinateAdd does not impact the behavior of the function as long as it can be determined that for all possible values of f, the argument $\texttt{pair}_1$ can be flattened. Although this example is contrived, it demonstrates that a logical data presentation (i.e., a grouping of coordinates into pairs) is not necessarily an optimal run-time data organization.

In general, all call sites of CoordinateAdd must be known to determine all possible candidates for f. However, such a determination about CoordinateAdd and f can be made only in the context of whole-program compilation, or if CoordinateAdd and f do not escape a function or module . If all candidates for f are not known, it may be the case that our assumptions about $\texttt{pair}_1$ uses are overly aggressive. Therefore, a compiler would need a method to alter $\texttt{pair}_1$'s representation. Without full knowledge about f, $\texttt{pair}_1$ cannot

[1] Unneeded representations can be removed by useless code elimination.

be flattened uniformly, instead a method to switch between boxed and unboxed representations is required. Alternatively, both representations can be passed to contexts in which tuple use information is not present to aggressively unbox. Our optimization, thus, benefits from whole program compilation in efficiency as un-needed tuple representations can be eliminated.

In this paper we explore a tuple unboxing algorithm for whole-program optimizing compilers. Our contributions are three-fold: (1) we present a formal semantics for the tuple flattening program transformation, (2) we provide a detailed proof of type safety and correctness over a simple intermediate language, and (3) we present a detailed study of the algorithm in the MLton Standard ML compiler, showing the approach is feasible even for large programs.

1.1 Related work

There exist many transformations that optimize the memory usage of a program [20, 21], unbox data [8], eliminate intermediate data structures instead of restructuring data representations [24], and split data-structure via arity raising [10]. Because these transformations remove overhead introduced by high-level abstractions, they have a potentially large performance benefit. Our transformation is distinguished from this family of optimizations, reducing overhead by eliminating memory indirections and unnecessary data stored within tuples. Tuples are a core element of functional language programs, serving to group and organize heterogeneous data objects. As data flows between functions, or even between different data structures, new intermediate tuples may be created. This is especially true of functional programming languages in which dataflow is expressed primarily through function invocation. Whenever an intermediate tuple is allocated, a cost is incurred to allocate the structure; subsequent costs accrue whenever its constituent elements are accessed.

Although potentially expensive, data boxing is an important compiler construct. Due to polymorphism the actual type of an object may not be discernible at compile time in most functional programs. Therefore, data representation decisions often cannot be inferred from the static type of an object. Compilers for ML-like languages typically represent structured data in boxed form [13] to avoid representational differences of data objects. Boxing enforces the invariant that all data structures have a uniform representation. Unfortunately, the consequence of boxing objects results in memory indirections to access data structure elements.

To limit memory access penalties, modern compilers discern opportunities to unbox data. Run-time type inspection [11], tag-based and type-directed unboxing [8], and coercion-based unboxing [14] are modern unboxing methods utilized by compilers and runtime systems. In run-time type inspection, the run-time representation of an object's type is stored within the program. Typing information is stored as extra arguments to polymorphic functions as well as extra components to structures defining abstract types. Types are inspected at run time to determine the location and size of values with polymorphic types. With tag-based unboxing, data structures are annotated with type information, effectively tagging the structure. Coercion-based unboxing inserts coercions when types are specialized so that monomorphic code utilizes unboxed representations while polymorphic code operates on boxed representations.

Unboxing implementations are not limited to optimizing data representations within functions, but are also applied to optimize data flow between functions. For example, unboxing can improve closure construction strategies through succinct data representations. TIL [19] and Flint [15], two middle ends for modern ML compilers, utilize unboxing [1, 4] for efficient closure representations [18]. Deforestation is a method for removing intermediate data structures [23]. Deforestation eliminates any unnecessary data structures passed between functions, but does not impact data structures that are not intermediate results [2, 25].

Type-directed unboxing [7, 8] relies on inducing data representations to fit their static type. Coercions are needed to handle cases of polymorphism where an argument of polymorphic type may be applied to actuals of different static types. Coercions locally change a tuple's representation from boxed to unboxed and vice-versa. They provide the correct version of a tuple in any given program context. However, coercions may introduce unnecessary overheads into a program. For example, calls to recursive functions wrapped with coercions result in an increase in size from linear to quadratic [9]. Any change of representation through a coercion requires rebuilding a tuple or extracting a tuple's components locally. There is no guarantee that a particular tuple will not be coerced many times (e.g., coercions may occur within a loop). In extreme cases, it is even possible for a tuple to constantly change representations in the absence of loops. For example, two mutually recursive functions can expect different representations of a particular argument they share in common.

One interesting version of type-directed unboxing is the method presented by Minamide and Garrigue [9], an extension of Leroy's [8] type-directed unboxing. A key contribution of Minamide and Garrigue's work revolves around keeping a reference boxed version of every function to provide assurances on the complexity of type-directed unboxing. This reference version is the original boxed version of the function, from which their algorithm is able to generate any specialized version with only one coercion. This approach guarantees that any particular representation is attainable through two coercions, one coercion to reach the reference function and one to specialize the function arbitrarily, avoiding potentially unbounded coercion chains.

Our approach differs from previous work in three key aspects. We eliminate duplicate selects on tuples by providing prefetched data. Secondly, our approach is coercion free, requiring no oscillations between representations. Thirdly, we specialize return points of functions, allowing us to specify different representations for different join points without generating different function representations.

1.2 Approach

Our flattening algorithm consists of three steps: labeling, transformation, and useless variable elimination. We define a transformation over an SSA intermediate language [6]. This decision allows us to avoid complexities that arise from polymorphic types, higher-order functions, etc., and permits us to achieve a measure of language independence. The SSA input program is first labeled by a simple transformation to *labeled* SSA. We define labeled SSA as the SSA language where all variables are annotated by the compiler with a label denoting whether or not a tuple should be flattened. Tuple variables that are to be unboxed will be labeled as F (flat). Tuple variables that are to remain boxed and non-tuple variables will be labeled as N (non-flat).

The labeled SSA program is then transformed back to SSA form through an application of our flattening transformation. The transformation phase of the flattening algorithm unboxes every tuple labeled flat, thus providing both a boxed and unboxed representation where necessary. If a tuple labeled flat is used in a flat manner, the unboxed representation will be chosen avoiding the memory indirection. Notice that a tuple labeled flat that is never used in a flat context will have its unboxed representation eliminated through useless code elimination. Likewise, any unnecessary boxed representations can also be eliminated through useless code elimination. Duplication of representations occurs only if a tuple is utilized in contexts which utilize both a boxed and unboxed representations.

Once a tuple is flattened, the association between the tuple and its flattened components is lost. Later uses of a tuple cannot be transformed without maintaining an association between a tuple and its prefetched components. To maintain the correlation between boxed and unboxed representations of a tuple, an unboxed representation is stored in a compile-time *tuple environment*. A tuple environment contains a simple one-to-one mapping between a tuple and a list of its flattened elements. An n-tuple labeled F will be followed by n bindings, one for each of its components. To accomplish this, n new variables are introduced:

$$x = (a, b, c) \rightarrow_{translation} x_1 = a,\ x_2 = b,\ x_3 = c\ x = (x_1, x_2, x_3) \tag{1}$$

The association between x and $\{x_1, x_2, x_3\}$ is added to the tuple environment. The original definition of the tuple is left unmodified and any use-site that expects a boxed tuple is free to utilize the original representation.

Our transformation utilizes useless code elimination as its third and final phase. Useless code elimination removes unnecessary tuple creations and tuple selects at any program point that requires only one representation. We rely on useless code elimination to minimize the number of duplicate representations needed throughout the program.

1.3 MLton

We evaluate the effectiveness of this flattening strategy in the context of MLton (Cejtin et al.), a whole-program optimizing compiler for SML. MLton compiles the full SML 97 language, including modules and functors. MLton supplies a host environment in which we can test our optimization and document any effects it may have on other optimization techniques.

MLton's approach to compilation can be summarized as whole-program optimization using a simply-typed first-order intermediate language. This approach is different from other functional language compilers and imposes significant constraints on the compiler, but yields many optimization opportunities not available with other approaches. We utilize MLton's compilation strategy to provide a flattening transformation that can be defined over a simple first order language.

There are numerous issues that arise when translating SML into a simply-typed IL. First, how does one represent SML modules and functors in a simply-typed IL, since these typically require much more complicated type systems? MLton's answer: defunctorize the program [3]. This transformation turns an SML program with modules into an equivalent one without modules by duplicating each functor at every application and eliminating structures by renaming variables. Second, how does one represent SML's polymorphic types and polymorphic functions in a simply-typed IL? MLton's answer: monomorphise the program [22]. This transformation eliminates polymorphism from an SML program by duplicating each polymorphic datatype and function at every type at which it is instantiated. Third, how does one represent SML's higher-order functions in a first-order IL? MLton's answer: defunctionalize the program [12]. This transformation replaces higher-order functions with data structures to represent them and first-order functions to apply them; the resulting IL is Static Single Assignment form.

Because each of the above transformations requires matching a functor, function definition, or type definition with all possible uses, MLton must be a whole-program compiler.

MLton's whole-program compilation strategy has a number of implications. Most importantly, MLton's use of defunctorization means that the placement of code in modules has *no effect* on performance. In fact, it has no effect on the generated code whatsoever. Modules are purely for the benefit of the programmer in structuring code. Also, because MLton duplicates functors at each use, no run-time penalty is incurred for abstracting a module into a functor. The benefits of monomorphisation are similar. Thus, with MLton, a programmer does not suffer the time and space penalties from an extra level of indirection in a list of doubles just because the compiler needs a uniform representation of lists. In MLton, whole-program control-flow analysis based on 0CFA [16] is employed early in the compilation process, immediately after defunctorization and monomorphisation, and well before any serious code motion or representation decisions are undertaken. Information computed by the analysis is used in the defunctionalization pass to introduce dispatches at call sites to the appropriate closure.

2 Semantics

We define our transformation over functional SSA, a variant of classical Static Single Assignment form. Functional SSA preserves the static singe assignment property, but differs from classic SSA in that all ϕ functions are eliminated [6]. Blocks in functional SSA correspond to ϕ functions; they represent join points within the program. Using the correspondence between ϕ functions and parameters to procedures whose call sites are known, all ϕ functions are replaced by multiple calls to one block.

In functional SSA, a program is a list of functions operating over values of either primitive or tuple type. One function in the program is distinguished as `main` denoting where program execution begins. A function is composed of a list of blocks followed by a transfer statement that shifts control to one of the function's blocks. Similarly a block is a list of statements, the last of which is a transfer. All functions and blocks are assumed to have unique identifiers. Statements are of the form $x : \tau = e$ and range over assignments, tuple creation, selects, and primitive operations. Primitive operations can constitute simple arithmetic or more complex actions dealing with arrays, tuples, vectors, and references.

Transfers represent a change of control, either locally through `Goto`'s or interprocedurally through `Call`'s. We distinguish explicitly between tail and non-tail calls in our grammar. Returns dispatch based on their target block, providing each block with a set of return values. Because we operate in a whole program compilation environment, all return points are explicitly known. The freedom to flatten every block uniquely in our transformation motivates this definition of `return`. Although the syntax distinguishes between specific return points, functional SSA passes the same arguments to each return. Our transformation relies on explicitly knowing all return points to allow per return point specialization of tuple representations. A transformed program passes different variables to each return target. The program's execution completes upon evaluation of a return with an empty call stack. We define a block identifier `halt` which defines the variables to be returned on program completion.

To illustrate functional SSA, consider the example SML program in Fig. 2 and its functional SSA representation in Fig. 3. The `main` function creates two vectors, calls `DotProduct`, extracts the second vector from the return value of `DotProduct`, and lastly calls `DotProduct` again providing a new vector (4.5, 5.5, 6.5). The `main` function returns the value returned by the second call to `DotProduct`. In functional SSA, `main` is written as a three block function. Block b_2 and `halt` represent the return point for the calls

$$
\begin{array}{rcl}
f, b, x \in \mathit{Name} & & \\
\iota \in \mathit{PrimType} & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \mid \ldots \\
\tau \in \mathit{Type} & ::= & \iota \mid (\tau_1 * \ldots * \tau_n) \mid \tau_1 \rightarrow \tau_2 \\
P \in \mathit{Prog} & ::= & \overline{F_n}; \mathtt{call}(\mathtt{main}, \mathtt{unit}) : \tau \\
F \in \mathit{Fun} & ::= & f = \lambda_f(\overline{x_n : \tau_n}) : \tau.\overline{B_n}; t \\
B \in \mathit{Block} & ::= & b = \lambda_b(\overline{x_n : \tau_n}) : \tau.\overline{S_n}; t \mid \mathtt{halt} \\
S \in \mathit{Statement} & ::= & x : \tau = e \\
e \in \mathit{Exp} & ::= & x \\
 & \mid & (\overline{x_n}) \\
 & \mid & \#i\ x \\
 & \mid & \mathit{Op}(op, \overline{x_n}) \\
t_i \in \mathit{Transfer} & ::= & \mathtt{call}(f, (\overline{x_n : \tau_n}), b) : \tau \\
 & \mid & \mathtt{call}(f, (\overline{x_n : \tau_n})) : \tau \\
 & \mid & \mathtt{goto}(b, (\overline{x_n : \tau_n})) : \tau \\
 & \mid & (\mathtt{if}\ e\ \mathtt{then}\ t_1 : \tau\ \mathtt{else}\ t_2 : \tau) : \tau \\
 & \mid & \mathtt{return}\ (b_1 \Rightarrow \overline{x_n : \tau_n} \mid \ldots \mid b_m \Rightarrow \overline{x_n : \tau_n}) : \tau
\end{array}
$$

Fig. 1 Functional SSA grammar

```
fun main() =
  let fun DotProduct(X:(real * real * real), Y:(real * real * real)) =
        let val x1 = #1 X
            val x2 = #2 X
            val x3 = #3 X
            val y1 = #1 Y
            val y2 = #2 Y
            val y3 = #3 Y
        in (x1 * y1 + x2 * y2 + x3 * y3, X, Y)
        end
      val vec = #3 DotProduct((1.0,2.0,3.0), (4.0,5.0,6.0))
  in DotProduct(vec, (4.5, 5.5, 6.5))
  end
```

Fig. 2 ML example

to `DotProduct`. The function `DotProduct` computes the dot product in R^3 where X and Y are vectors in real space. Constituent components of the vectors X and Y (represented as tuples in the function) are multiplied together and then summed. The function returns the dot product of X and Y, as well as the original vectors. This function uses tuples X and Y in both flat and non-flat ways. The functional SSA representation of `DotProduct` remains essentially the same as the ML function, and consists of a single function call, and one block. Notice the return in `DotProduct` distinguishes between its two possible return points, b_2 and `halt`.

2.1 Functional SSA evaluation and type checking

The domains and evaluation of functional SSA programs is given in Figs. 4, 5, and 6. We write $x_1, \ldots, x_n$ to denote a sequence of n elements, $\overline{x_n}$ as a shorthand for a sequence of n elements, $\overline{x}$ as a sequence of zero or more elements, ϵ as the empty sequence, and $\overline{x}; \overline{y}$ as the concatenation of two sequences. We define a reduction relation $\rightarrow_E$, which reduces a functional SSA program to a value. We define ρ as an environment mapping variables to values and $\rho_\perp$ as the empty environment. We write $\rho_\perp \vdash \overline{F_n}; \mathtt{call}(\mathtt{main}, \mathtt{unit}) : \tau \rightarrow_E v$ to denote the evaluation of program P to value v. We also write $\rho \vdash e \rightarrow_E v$, if the expression e evaluates to the value v in the environment ρ and $\rho \vdash S \rightarrow_B \rho'$, if the evaluation

```
main = $\lambda_f(x_0)$.
 $b_1$ = $\lambda_b()$.
   $x_1$ = 1.0
   $x_2$ = 2.0
   $x_3$ = 3.0
   $x_4$ = ($x_1$, $x_2$, $x_3$)
   $x_5$ = 4.0
   $x_6$ = 5.0
   $x_7$ = 6.0
   $x_8$ = ($x_5$, $x_6$, $x_7$)
   call(DotProduct, ($x_4$, $x_8$), $b_2$);
 $b_2$ = $\lambda_b(x_9)$.
   $x_{10}$ = #3 $x_9$
   $x_{11}$ = 4.5
   $x_{12}$ = 5.5
   $x_{13}$ = 6.5
   $x_{14}$ = ($x_{11}$, $x_{12}$, $x_{13}$)
   call(DotProduct, ($x_{10}$, $x_{14}$));
 goto($b_1$);

DotProduct = $\lambda_f$(X:(real*real*real), Y:(real*real*real)).
 $b_3$ = $\lambda_b()$.
    $x'_1$ = #1 X
    $x'_2$ = #2 X
    $x'_3$ = #3 X
    $y_1$ = #1 Y
    $y_2$ = #2 Y
    $y_3$ = #3 Y
    $z_1$ = Op(*, $x'_1$, $y_1$)
    $z_2$ = Op(*, $x'_2$, $y_2$)
    $z_3$ = Op(*, $x'_3$, $y_3$)
    $a_1$ = Op(+, $z_1$, $z_2$, $z_3$)
    $a_2$ = ($a_1$, $Y$, $X$)
    return($b_2$ => $a_2$ | halt => $a_2$);
  goto($b_3$);

call(main, unit)
```

Fig. 3 Functional SSA example

of statement S results in environment ρ'. Transfers operate over a stack of target block and environment pairs for non-tail calls. We write $\rho, Bs \vdash t \rightarrow_T v$, if the transfer t evaluates with the stack Bs to yield value v. For convenience we will switch between the names of blocks and their concrete representations. The program transfer `halt` will return its value without transferring control to another block, thus terminating the program.

We assume the program is annotated with types and all function and block names are unique. To reason about type safety, we define a typing relation in Figs. 7 and 8. The definition is standard; we write $\Gamma \vdash e : \tau$, if expression e has type τ in the type environment Γ. We also assume all primitive operations have types bound in $\Gamma_\perp$, the initial typing environment. Type checking begins by extending the initial typing environment with function names mapped to their types. Similarly we extend the typing environment when typechecking a function with the blocks that compose the function. The transfer that begins the functions (or blocks) evaluation is a local transfer. Any transfers within the body of the function's (or block's) can transfer out of the function. The function's return type is defined by its return transfers. Each transfer executes a transition of $\rightarrow$ type. The type of the transfer is the return

$$\begin{aligned}
\pi &\in \mathit{Term} = \mathit{Prog} \mid \mathit{Fun} \mid \mathit{Block} \mid \mathit{Statement} \mid \mathit{Exp} \mid \mathit{Transfer} \mid \mathit{Var}\\
\phi_e &\in \mathit{EvalTerm} = \mathit{Prog} \mid \mathit{Exp}\\
\rho &\in \mathit{Env} = \mathit{Var} \to \mathit{Value}\\
\Gamma &\in \mathit{TypeEnv} = \mathit{Var} \to \mathit{Type}\\
v &\in \mathit{Value} = 0 \mid 1 \mid ... \mid \mathit{true} \mid \mathit{false} \mid (v_1, ..., v_n) \mid \mathtt{unit}\\
\mathit{op} &\in \mathit{PrimOp} = + \mid - \mid * \mid \mathtt{ref} \mid \mathtt{deref} \mid ...\\
\mathit{Bs} &\in \mathit{Stack} = (\mathit{Name} \times \mathit{Env})^*\\
\to_E &\in \mathit{Eval} = \mathit{Env} \times \mathit{EvalTerm} \to \mathit{Value}\\
\to_T &\in \mathit{EvalT} = \mathit{Prog} \times \mathit{Env} \times \mathit{Stack} \times \mathit{Transfer} \to \mathit{Value}\\
\to_B &\in \mathit{Bind} = \mathit{Env} \times \mathit{Statements} \to \mathit{Env}\\
\aleph &\in \mathit{PrimApp} = \mathit{Op} \times \mathit{Values} \to \mathit{Value}
\end{aligned}$$

Fig. 4 Functional SSA evaluation domain

(Statement)

$$\frac{\rho \vdash e \to_E v}{\rho \vdash x : \tau = e \to_B \rho[x \mapsto v]} \tag{2}$$

$$\frac{\begin{array}{c}\rho_0 = \rho\\ \rho_{i-1} \vdash s_i \to_B \rho_i \quad i = 1, ..., n\end{array}}{\rho \vdash \overline{s_n} \to_B \rho_n} \tag{3}$$

(Exp)

$$\frac{\rho(x) = v}{\rho \vdash x \to_E v} \tag{4}$$

$$\frac{\rho(x_i) = v_i \quad i = 1, ..., n}{\rho \vdash (\overline{x_n}) \to_E (\overline{v_n})} \tag{5}$$

$$\frac{\begin{array}{c}\rho(x_i) = v_i \quad i = 1, ..., n\\ \aleph(\mathit{op}, \overline{v_n}) = v\end{array}}{\rho \vdash \mathit{Op}(\mathit{op}; \overline{x_n}) \to_E v} \tag{6}$$

$$\frac{\rho(y) = (\overline{v_n}) \quad i = 1, ..., n}{\rho \vdash \#i\ y \to_E v_i} \tag{7}$$

Fig. 5 Functional SSA evaluation

type of the function (or block) that it executes. Notice that a return is equivalent to a goto and an if then else will also either execute a call or a goto.

Lemma 1

(1) *if* $\Gamma \vdash x : \tau = e : \tau$ *then* $FV(e) \in Dom(\Gamma)$
(2) *if* $\Gamma \vdash t : \tau$ *then* $FV(t) \in Dom(\Gamma)$.

Proof Simple induction on typing derivations. □

Lemma 2

(1) *if* $\rho \vdash x : \tau = e \to_B \rho[x \mapsto v]$ *then* $FreeVars(e) \in Dom(\rho)$
(2) *if* $\rho, Bs \vdash t \to_t v$ *then* $FreeVars(t) \in Dom(\rho)$.

Proof Simple induction on evaluation derivations. □

(Transfer)

$$\frac{\begin{array}{c} b = \lambda_b(\overline{x_n : \tau_n}) : \tau'.\overline{S}; t \in P \\ \rho[x_i \mapsto \rho(y_i)] \vdash \overline{S} \rightarrow_B \rho' \quad i = 1, ..., n \\ P, \rho', Bs \vdash t : \tau \rightarrow_T v \end{array}}{P, \rho, Bs \vdash \texttt{goto}(b, (\overline{y_n})) : \tau \rightarrow_T v} \tag{8}$$

$$\frac{\begin{array}{c} f = \lambda_f(\overline{x_n : \tau_n}) : \tau'.\overline{B}; t \in P \\ P, \rho_\perp[x_i \mapsto \rho(y_i)], (b, \rho); Bs \vdash t : \tau \rightarrow_T v \quad i = 1, ..., n \end{array}}{P, \rho, Bs \vdash \texttt{call}(f, (\overline{y_n}), b) : \tau \rightarrow_T v} \tag{9}$$

$$\frac{\begin{array}{c} f = \lambda_f(\overline{x_n : \tau_n}) : \tau'.\overline{B}; t \in P \\ P, \rho_\perp[x_i \mapsto \rho(y_i)], Bs \vdash t : \tau \rightarrow_T v \quad i = 1, ..., n \end{array}}{P, \rho, Bs \vdash \texttt{call}(f, (\overline{y_n})) : \tau \rightarrow_T v} \tag{10}$$

$$\frac{\begin{array}{c} \rho \vdash e \rightarrow_E \mathit{true} \\ P, \rho, Bs \vdash t_1 : \tau \rightarrow_T v \end{array}}{P, \rho, Bs \vdash (\texttt{if}\ e\ \texttt{then}\ t_1 : \tau\ \texttt{else}\ t_2 : \tau) : \tau \rightarrow_T v} \tag{11}$$

$$\frac{\begin{array}{c} \rho \vdash e \rightarrow_E \mathit{false} \\ P, \rho, Bs \vdash t_2 : \tau \rightarrow_T v \end{array}}{P, \rho, Bs \vdash (\texttt{if}\ e\ \texttt{then}\ t_1 : \tau\ \texttt{else}\ t_2 : \tau) : \tau \rightarrow_T v} \tag{12}$$

$$\frac{\begin{array}{c} Bs = (b, \rho'); Bs' \\ \rho(x_i) = v_i \quad i = 1, ..., n \\ P, \rho', Bs' \vdash \texttt{goto}(b, (\overline{v_n})) : \tau' \rightarrow_T v \end{array}}{P, \rho, Bs \vdash \texttt{return}(b \Rightarrow x_1 : \tau_1, ..., x_n : \tau_n \mid \ldots) : \tau \rightarrow_T v} \tag{13}$$

$$\frac{\begin{array}{c} P \in \mathit{Prog} \quad P = \overline{F_n}; \texttt{call}(\texttt{main}, \texttt{unit}) : \tau \\ P, \rho_\perp, \epsilon \vdash \texttt{call}(\texttt{main}, \texttt{unit}) : \tau \rightarrow_T v \end{array}}{\rho_\perp \vdash \overline{F_n}; \texttt{call}(\texttt{main}, \texttt{unit}) : \tau \rightarrow_E v} \tag{14}$$

Fig. 6 Functional SSA evaluation (continued)

3 Transformation

3.1 Labeling

Our transformation supports any labeling, allowing compilers to modify their flattening strategy to their specific needs. We use labels N and F, where N denotes *non-flat* and F denotes *flat*, to annotate SSA variables. A variable x that has been annotated with a labeling is referred to as a *labeled variable*. We write $L(x)$ to denote the label of x. For uniformity, we mark every non-tuple variable with label *non-flat*. In SSA, a tuple as well as its constituent component must be bound to variables. These variables can have any arbitrary labeling. Therefore, to ensure that the flat representation is available for a given tuple we emit new variables, representing each of the tuples components, and label them *flat*. Any tuple definition labeled *non-flat* is boxed and remains so for the entirety of the program.

During whole program compilation, all uses of a variable are known and thus a consistent labeling can be assured. We constrain that a dispatch statement in a `return` for some target block b to be labeled exactly as the arguments for b. Since all return points are explicitly known, each can have a unique labeling based on the formals the return point expects. All arguments to `calls` and `gotos` are labeled based on what their target function or block expects (see Fig. 10).

To store a correspondence between the nesting structure of the tuple and the emitted variables we introduce a tuple environment δ. For each nesting level of each tuple component we

$$\frac{\Gamma[x_i \mapsto \tau_i] \vdash \overline{S}; t : \tau \quad i = 1, ..., n}{\Gamma \vdash (b = \lambda_b(\overline{x_n : \tau_n}) : \tau.\overline{S}; t) : (\tau_1 * ... * \tau_n) \rightarrow \tau} \tag{15}$$

$$\frac{\begin{array}{c} \Gamma[x_i \mapsto \tau_i] \vdash t : \tau \quad i = 1, ..., n \\ \overline{B} = b_1, ..., b_n \\ b_i = \lambda_b(\overline{x_m : \tau_m}) : \tau_i.\overline{S}; t \quad \tau_i' = (\tau_1 * ... * \tau_m) \rightarrow \tau_i \quad i = 1, ..., n \\ \Gamma' = \Gamma[b_i \mapsto \tau_i'] \quad \Gamma' \vdash b_k : \tau_k \quad i, k = 1, ..., n \end{array}}{\Gamma \vdash (f = \lambda_f(\overline{x_n : \tau_n}) : \tau.\overline{B}; t) : (\tau_1 * ... * \tau_n) \rightarrow \tau} \tag{16}$$

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (\texttt{if}\ e\ \texttt{then}\ t_1 : \tau\ \texttt{else}\ t_2 : \tau) : \tau} \tag{17}$$

$$\frac{\begin{array}{c} \Gamma(b_i) = (\tau_1 * ... * \tau_n) \rightarrow \tau \quad i = 1, ..., j \\ \overline{x_n : \tau_n} = x_1 : \tau_1, ..., x_n : \tau_n \quad \Gamma(x_i) = \tau_i \quad i = 1, ..., n \end{array}}{\Gamma \vdash \texttt{return}(b_1 \Rightarrow \overline{x_n : \tau_n} \mid \ldots \mid b_j \Rightarrow \overline{x_n : \tau_n}) : \tau} \tag{18}$$

$$\frac{\begin{array}{c} \Gamma(x_i) = \tau_i \quad i = 1, ..., n \\ \Gamma(b) = (\tau_1 * ... * \tau_n) \rightarrow \tau \end{array}}{\Gamma \vdash \texttt{goto}(b, (\overline{x_n})) : \tau} \tag{19}$$

$$\frac{\begin{array}{c} \Gamma(x_i) = \tau_i \quad i = 1, ..., n \\ \Gamma(f) = (\tau_1 * ... * \tau_n) \rightarrow \tau \end{array}}{\Gamma \vdash \texttt{call}(f, (\overline{x_n})) : \tau} \tag{20}$$

$$\frac{\begin{array}{c} \Gamma(x_i) = \tau_i \quad i = 1, ..., n \\ \Gamma(f) = (\tau_1 * ... * \tau_n) \rightarrow \tau \end{array}}{\Gamma \vdash \texttt{call}(f, (\overline{x_n}), b) : \tau} \tag{21}$$

$$\frac{\begin{array}{c} f_i = \lambda_f(\overline{x_m : \tau_m}) : \tau_i.\overline{B_j}; t \quad \tau_i' = (\tau_1 * ... * \tau_m) \rightarrow \tau_i \quad i = 1, ..., n \\ \Gamma' = \Gamma_\perp[f_i \mapsto \tau_i'] \quad \Gamma' \vdash F_k : \tau_k \quad i = 1, ..., n \quad k = 1, ..., n \\ \Gamma' \vdash \texttt{call}(\texttt{main}, \texttt{unit}) : \tau_j \quad i = 1, ..., n \end{array}}{\Gamma_\perp \vdash \overline{F_n}; \texttt{call}(\texttt{main}, \texttt{unit}) : \tau_j} \tag{22}$$

Fig. 7 Functional SSA typing rules

$$\frac{\begin{array}{c} \Gamma(y_i) = \tau_i \quad i = 1, ..., n \\ \tau = (\tau_1 * ... * \tau_n) \end{array}}{\Gamma \vdash x : \tau = (\overline{y_n}) : (\tau_1 * ... * \tau_n)} \tag{23}$$

$$\frac{\begin{array}{c} \Gamma(y) = (\tau_1 * ... * \tau_n) \quad i = 1, ..., n \\ \tau = \tau_i \end{array}}{\Gamma \vdash x : \tau = \#i\ y : \tau_i} \tag{24}$$

$$\frac{\begin{array}{c} \Gamma(op) = (\tau_1 * ... * \tau_n) \rightarrow \tau \\ \Gamma(y_i) = \tau_i \quad i = 1, ..., n \end{array}}{\Gamma \vdash x : \tau = Op(op; \overline{y_n}) : \tau} \tag{25}$$

$$\frac{\begin{array}{c} S_i = (x_i : \tau_i = e_i) \quad i = 1, ..., n \\ \Gamma \vdash S_1 : \tau_1 \ldots \Gamma[x_i \mapsto \tau_i] \vdash S_n : \tau_n \quad i = 1, ..., n-1 \\ \Gamma[x_j \mapsto \tau_j] \vdash t : \tau \quad j = 1, ..., n \end{array}}{\Gamma \vdash \overline{S_n}; t : \tau} \tag{26}$$

Fig. 8 Functional SSA typing rules (continued)

emit fresh variables and create a mapping in the tuple environment. For example, an argument x with the following type, `(int * int)*(int * int)`, would be represented as $\delta[x \rightarrow_\delta x_1, x_2; x_1 \rightarrow_\delta x_{11}, x_{12}; x_2 \rightarrow_\delta x_{21}, x_{22}]$ where $x_1, x_2, x_{11}, x_{12}, x_{21}, x_{22}$ are fresh. The

Fig. 9 Tuple environment domain

$$\begin{aligned} L &\in \mathit{Labeling} = \mathit{Var} \rightarrow \mathit{Label} \\ \delta &\in \mathit{TupleEnv} = \mathit{Var} \rightarrow \mathit{TypedVars} \\ \rightarrow_{\delta}, \rightharpoonup_{\delta} &\in \mathit{Fold} = \mathit{Labeling} \times \mathit{TupleEnv} \times \mathit{Terms} \\ &\quad \rightarrow \mathit{Labeling} \times \mathit{TupleEnv} \end{aligned}$$

$$\frac{\begin{array}{c} f = \lambda_f(\overline{x_n : \tau_n}) : \tau'.\overline{B}; t \\ L \vdash t \\ L(y_i) = L(x_i) \quad i = 1, \ldots, n \end{array}}{L \vdash \mathtt{call}(f, (\overline{y_n}), b)} \tag{27}$$

$$\frac{\begin{array}{c} f = \lambda_f(\overline{x_n : \tau_n}) : \tau'.\overline{B}; t \\ L \vdash t \\ L(y_i) = L(x_i) \quad i = 1, \ldots, n \end{array}}{L \vdash \mathtt{call}(f, (\overline{y_n}))} \tag{28}$$

$$\frac{\begin{array}{c} b = \lambda_b(\overline{x_n : \tau_n}) : \tau'.\overline{S}; t \\ L(y_i) = L(x_i) \quad i = 1, \ldots, n \end{array}}{L \vdash \mathtt{goto}(b, (\overline{y_n}))} \tag{29}$$

$$\frac{L \vdash \mathtt{goto}(b_i, (\overline{x_n})) \quad i = 1, \ldots, m}{L \vdash \mathtt{return}(b_1 \Rightarrow \overline{x_n : \tau_n} \mid \ldots \mid b_m \Rightarrow \overline{x_n : \tau_n})} \tag{30}$$

$$\frac{L \vdash t_1 \quad L \vdash t_2}{L \vdash (\mathtt{if}\ e\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2)} \tag{31}$$

Fig. 10 Labeling constraints: types annotations omitted where unnecessary

fresh variables x_1 and x_2 would both be labeled as *flat* because they are of tuple type and the remaining fresh variables x_{11}, x_{12}, x_{21}, and x_{22} would be labeled *non-flat* because they are of primitive type. In order to keep our rules simple, we present the transformation in two phases: the first builds a tuple environment and the second injects the fresh variables stored in the tuple environment. In the first phase, we build the tuple environment through the relations $\rightarrow_{\delta}$ and $\rightharpoonup$ starting from the empty tuple environment $\delta_{\perp}$, presented in Figs. 9 and 11. The tuple environment creation generates freshly labeled variables.

We write $x_1, \ldots, x_n$ *fresh* to denote the creation of n new labeled variables corresponding to components of a tuple; the types of the variables are defined by the types of the tuple's components. The fresh variable is labeled *flat* if it is of tuple type and *non-flat* otherwise.

Since many different possibilities exist for labeling a program, it is difficult to uniquely determine or infer an optimal labeling. Experimental results indicate that it is necessary to adapt a labeling to a particular compiler and application. Optimizations can affect which data should be unboxed and which should remain boxed. Applications themselves can also affect the utility of a labeling. For instance, user-declared tuples of thousands of elements should probably not be flattened, regardless of the efficiency of the transformation.

3.2 Flattening

Our flattening rules are defined by a transformation relation (see Fig. 12) $\rightharpoondown_T$ operating over a tuple environment, a labeling L, and π, the current term being transformed. Given a particular tuple environment δ, we write $\pi \rightharpoondown \psi$ if the transformation of term π yields a sequence of terms ψ. To transform our program under $\rightharpoondown$ we introduce an auxiliary function $\rightharpoonup_F$, shown in Fig. 13, that performs a fold over $\rightharpoondown$. The definitions of $\rightharpoondown$ and $\rightharpoonup_F$ are mutually recursive, but restricted as follows: if $\pi \rightharpoondown \pi'$ and π_s is a subterm of π then $\rightharpoonup_F$

$$\frac{}{L, \delta \vdash \epsilon \rightharpoonup_{\delta} L, \delta} \tag{32}$$

$$\frac{\begin{array}{c} L, \delta \vdash \pi \rightarrow_{\delta} L', \delta' \\ L', \delta' \vdash \pi s \rightharpoonup_{\delta} L'', \delta'' \end{array}}{L, \delta \vdash \pi; \pi s \rightharpoonup_{\delta} L'', \delta''} \tag{33}$$

$$\frac{\begin{array}{c} L(x) = F \quad x_1, ..., x_n \textit{ fresh} \\ L[x_1 \mapsto F], \delta \vdash x_1 : \tau_1 \rightarrow_{\delta} L_1, \delta_1 \\ ... \\ L_{n-1}[x_n \mapsto F], \delta_{n-1} \vdash x_n : \tau_n \rightarrow_{\delta} L_n, \delta_n \quad i = 1, ..., n \end{array}}{L, \delta \vdash x : (\tau_1 * ... * \tau_n) \rightarrow_{\delta} L_n, \delta_n[x \mapsto \overline{x_n : \tau_n}]} \tag{34}$$

$$\frac{\begin{array}{c} L(x) = F \quad x_1, ..., x_n \textit{ fresh} \\ L[x_1 \mapsto N], \delta \vdash x_1 : \tau_1 \rightarrow_{\delta} L_1, \delta_1 \\ ... \\ L_{n-1}[x_n \mapsto N], \delta_{n-1} \vdash x_n : \tau_n \rightarrow_{\delta} L_n, \delta_n \quad i = 1, ..., n \end{array}}{L, \delta \vdash x : (\tau_1 * ... * \tau_n) = e \rightarrow_{\delta} L_n, \delta_n[x \mapsto \overline{x_n : \tau_n}]} \tag{35}$$

$$\frac{L(x) = N}{L, \delta \vdash x : \tau \rightarrow_{\delta} L, \delta} \tag{36}$$

$$\frac{}{L, \delta \vdash x : \iota \rightarrow_{\delta} L, \delta} \tag{37}$$

$$\frac{L(x) = N}{L, \delta \vdash x : \tau = e \rightarrow_{\delta} L, \delta} \tag{38}$$

$$\frac{L, \delta_{\perp} \vdash \overline{F_n} \rightharpoonup_{\delta} L', \delta}{L, \delta_{\perp} \vdash \overline{F_n}; \mathtt{call}(\mathtt{main}, \mathtt{unit}) \rightarrow_{\delta} L', \delta} \tag{39}$$

$$\frac{\begin{array}{c} L, \delta \vdash \overline{x_n : \tau_n} \rightharpoonup_{\delta} L', \delta' \\ L', \delta' \vdash \overline{B} \rightharpoonup_{\delta} L'', \delta'' \end{array}}{L, \delta \vdash f = \lambda_f(\overline{x_n : \tau_n}) : \tau.\overline{B}; t \rightarrow_{\delta} L'', \delta''} \tag{40}$$

$$\frac{\begin{array}{c} L, \delta \vdash \overline{x_n : \tau_n} \rightharpoonup_{\delta} L', \delta' \\ L', \delta' \vdash \overline{S} \rightharpoonup_{\delta} L'', \delta'' \end{array}}{L, \delta \vdash b = \lambda_b(\overline{x_n : \tau_n}) : \tau.\overline{S}; t \rightarrow_{\delta} L'', \delta''} \tag{41}$$

Fig. 11 Building a tuple environment

Fig. 12 Functional SSA transformation rules domain

$$\begin{array}{rl} \psi & \in \textit{Terms} \\ \rightharpoonup_F, \rightharpoondown & \in \textit{Transform} = \textit{Labeling} \times \textit{TupleEnv} \times \textit{Term} \rightarrow \textit{Terms} \\ \rightsquigarrow_F & \in \textit{FoldSel} = \textit{TupleEnv} \times \textit{Terms} \rightarrow \textit{Terms} \\ \rightsquigarrow_S & \in \textit{Select} = \textit{TupleEnv} \times \textit{Term} \rightarrow \textit{Terms} \end{array}$$

can be applied to π_s. Because only subterms will be evaluated with $\rightharpoonup_F$, the recursion is bounded. We define ψ as a short hand notion for a sequence of π terms.

The rules for *Prog*, *Fun*, and *Block* in Fig. 15 are similar. Each rule progresses over a list of terms via $\rightharpoonup_F$, building up ψ. The rule for *Prog* operates on a list of functions, applying $\rightharpoondown$ to each function. The rules for function and block are slightly more complex since both can take arguments, but they are similar in structure to the rule for *Prog*.

The argument rule retrieves an argument mapped in the tuple environment (see Fig. 16). Since all arguments labeled F (*flat*) will contain appropriate mappings based on a labeling pass, it is enough to retrieve fresh variables from the tuple environment. Any argument labeled N(*non-flat*) does not contain a mapping and therefore cannot be flattened. The map-

$$\frac{}{\delta, \epsilon \rightharpoonup_F \epsilon} \tag{42}$$

$$\frac{\begin{array}{c} L, \delta, \pi \rightharpoondown \psi \\ L, \delta, \pi s \rightharpoonup_F \psi' \end{array}}{L, \delta, \pi; \pi s \rightharpoonup_F \psi; \psi'} \tag{43}$$

$$\frac{\begin{array}{c} P \in \mathit{Prog} \quad P = \overline{F_n}; \texttt{call}(\texttt{main}, \texttt{unit}) : \tau \\ L, \delta_\perp \vdash P \rightarrow_\delta L', \delta \\ L', \delta, P \rightharpoondown \overline{F'_n}, \texttt{call}(\texttt{main}, \texttt{unit}) : \tau \end{array}}{L, \delta_\perp, \overline{F_n}; \texttt{call}(\texttt{main}, \texttt{unit}) : \tau \rightharpoondown \overline{F'_n}; \texttt{call}(\texttt{main}, \texttt{unit}) : \tau} \tag{44}$$

Fig. 13 Functional SSA transformation helper functions

$$\frac{\begin{array}{c} \delta, x_1 : \tau_1 \leadsto_S \psi \\ \delta, x_2 : \tau_2, \ldots, x_n : \tau_n \leadsto_F \psi' \end{array}}{\delta, \overline{x_n : \tau_n} \leadsto_F \psi; \psi'} \tag{45}$$

$$\frac{\begin{array}{c} \delta(x) = \overline{x_n : \tau_n} \\ \psi = x_1 : \tau_1 = \#1\ x, \ldots, x_n : \tau_n = \#n\ x \\ \delta, \overline{x_n : \tau_n} \leadsto_F \psi' \end{array}}{\delta, x : \tau \leadsto_S \psi; \psi'} \tag{46}$$

$$\frac{}{\delta, x : \iota \leadsto_S \epsilon} \tag{47}$$

Fig. 14 Selecting out a tuple's structure

(Prog)

$$\frac{L, \delta, \overline{F_n} \rightharpoonup_F \psi}{L, \delta, \overline{F_n}; \texttt{call}(\texttt{main}, \texttt{unit}) : \tau \rightharpoondown \psi; \texttt{call}(\texttt{main}, \texttt{unit}) : \tau} \tag{48}$$

(Fun)

$$\frac{\begin{array}{c} L, \delta, \overline{x_n : \tau_n} \rightharpoonup_F \psi \\ L, \delta, \overline{B} \rightharpoonup_F \psi' \\ L, \delta, t : \tau \rightharpoonup_F t' : \tau' \end{array}}{L, \delta, f = \lambda_f(\overline{x_n : \tau_n}) : \tau.\overline{B}; t \rightharpoondown f = \lambda_f(\psi) : \tau'.\psi'; t} \tag{49}$$

(Block)

$$\frac{\begin{array}{c} L, \delta, \overline{x_n : \tau_n} \rightharpoonup_F \psi \\ L, \delta, \overline{S} \rightharpoonup_F \psi' \\ L, \delta, t : \tau \rightharpoonup_F t' : \tau' \end{array}}{L, \delta, b = \lambda_b(\overline{x_n : \tau_n}) : \tau.\overline{S}; t \rightharpoondown b = \lambda_b(\psi) : \tau'.\psi'; t} \tag{50}$$

Fig. 15 Functional SSA transformation rules—program, function, block

pings in the tuple environment represent all nesting levels of the tuple, providing a complete flat representation of the tuple.

There are two types of bindings that must to be considered. If the left hand side is labeled flat the tuple's structure is selected out. If the left hand side is labeled non-flat, the statement is unchanged. The relation $\leadsto_S$, defined in Fig. 14, provides a full flat representation. The rules for tuples in Fig. 18 and primitive operations in Fig. 19 are similar to the assignment rules. Primitive operations, shown in Fig. 19, are not transformed under $\rightharpoondown$. Instead,

(Argument)

$$\frac{\begin{array}{c} L(x) = F \\ \delta(x) = \overline{x_n : \tau_n} \\ L, \delta, x_i : \tau_i \rightharpoondown \psi_i \quad i = 1, ..., n \end{array}}{L, \delta, x : \tau \rightharpoondown x : \tau; \overline{\psi_n}} \tag{51}$$

$$\frac{L(x) = N}{L, \delta, x : \tau \rightharpoondown x : \tau} \tag{52}$$

Fig. 16 Functional SSA transformation rules—argument

(Assignment)

$$\frac{L(x) = N}{L, \delta, x : \tau = y \rightharpoondown x : \tau = y} \tag{53}$$

$$\frac{\begin{array}{c} L(x) = F \\ \delta, x : \tau \rightsquigarrow_S \psi \end{array}}{L, \delta, x : \tau = y \rightharpoondown x : \tau = y; \psi} \tag{54}$$

Fig. 17 Functional SSA transformation rules—assignment

(Tuple)

$$\frac{\begin{array}{c} L(x) = F \\ \delta, x : \tau \rightsquigarrow_S \psi \end{array}}{L, \delta, x : \tau = (\overline{y_n}) \rightharpoondown x : \tau = (\overline{y_n}); \psi} \tag{55}$$

$$\frac{L(x) = N}{L, \delta, x : \tau = (\overline{y_n}) \rightharpoondown x : \tau = (\overline{y_n})} \tag{56}$$

Fig. 18 Functional SSA transformation rules—tuple

(Op)

$$\frac{\begin{array}{c} L(x) = F \\ \delta, x : \tau \rightsquigarrow_S \psi \end{array}}{L, \delta, x : \tau = Op(op, \overline{y_n}) \rightharpoondown x : \tau = Op(op, \overline{y_n}); \psi} \tag{57}$$

$$\frac{L(x) = N}{L, \delta, x : \tau = Op(op, \overline{y_n}) \rightharpoondown x : \tau = Op(op, \overline{y_n})} \tag{58}$$

Fig. 19 Functional SSA transformation rules—primitive

arguments to primitives are always passed boxed. Primitives can potentially introduce new tuples into the program. If a primitive introduces a new tuple, we flatten the tuple based on its labeling.

As shown in Fig. 20, the translation of terms of the form $x : \tau = \#i\ y$, revolves around exposing one nesting level of a tuple. If the target tuple y is labeled F (*flat*), we check the tuple environment for y's constituent components. Whenever there exists a select on a flat tuple, our transformation eliminates the memory indirection with a simple assignment statement, $x : \tau = y_i$, where y_i corresponds to the statement $y_i : \tau = \#i\ y$ that has been

(Select)

$$\frac{\begin{array}{c} L(x) = F \;\; L(y) = F \\ \delta(y) = (\overline{y_n : \tau_n}) \\ \delta, x : \tau \rightsquigarrow_S \psi \end{array}}{L, \delta, x : \tau = \#i\ y \rightharpoondown x : \tau = y_i ; \psi} \tag{59}$$

$$\frac{\begin{array}{c} L(x) = N \;\; L(y) = F \\ \delta(y) = (\overline{y_n : \tau_n}) \end{array}}{L, \delta, x : \tau = \#i\ y \rightharpoondown x : \tau = y_i} \tag{60}$$

$$\frac{\begin{array}{c} L(x) = F \;\; L(y) = N \\ \delta, x : \tau \rightsquigarrow_S \psi \end{array}}{L, \delta, x : \tau = \#i\ y \rightharpoondown x : \tau = \#i\ y ; \psi} \tag{61}$$

$$\frac{L(x) = N \;\; L(y) = N}{L, \delta, x : \tau = \#i\ y \rightharpoondown x : \tau = \#i\ y} \tag{62}$$

Fig. 20 Functional SSA transformation rules—select

(Goto)

$$\frac{\begin{array}{c} L, \delta, \overline{x_n : \tau_n} \rightharpoonup_F \psi \\ L, \delta, b = \lambda_b(\overline{x_n : \tau_n}) : \tau.\overline{S}; t \rightharpoondown b = \lambda_b(\psi') : \tau'.\psi''; t' \end{array}}{L, \delta, \texttt{goto}(b, (\overline{x_n : \tau_n})) : \tau \rightharpoondown \texttt{goto}(b, (\psi)) : \tau'} \tag{63}$$

(Call)

$$\frac{\begin{array}{c} L, \delta, \overline{x_n : \tau_n} \rightharpoonup_F \psi \\ L, \delta, f = \lambda_f(\overline{x_n : \tau_n}) : \tau.\overline{B}; t \rightharpoondown f = \lambda_f(\psi') : \tau'.\psi''; t' \end{array}}{L, \delta, \texttt{call}(f, (\overline{x_n : \tau_n}), b) : \tau \rightharpoondown \texttt{call}(f, (\psi), b) : \tau'} \tag{64}$$

Fig. 21 Functional SSA transformation rules—transfer

(If Then Else)

$$\frac{\begin{array}{c} L, \delta, t_1 : \tau \rightharpoonup_F t_1' : \tau' \\ L, \delta, t_2 : \tau \rightharpoonup_F t_2' : \tau' \\ \psi = (\texttt{if}\ e\ \texttt{then}\ t_1' : \tau'\ \texttt{else}\ t_2' : \tau') : \tau' \end{array}}{L, \delta, (\texttt{if}\ e\ \texttt{then}\ t_1 : \tau\ \texttt{else}\ t_2 : \tau) : \tau \rightharpoondown \psi} \tag{65}$$

(Return)

$$\frac{\begin{array}{c} L, \delta, \overline{x_n : \tau_n} \rightharpoonup_F \psi_i \quad i = 1, ..., j \\ \psi_i = \overline{x_m : \tau_m} \quad \tau_i = (\tau_1 * ... * \tau_m) \quad i = 1, ..., j \\ \tau' = \tau_l + ... + \tau_k \quad \tau_l \neq \tau_k \quad l, k \in 1, ..., j \\ \psi = \texttt{return}(b_1 \Rightarrow \psi_1 \mid \ldots \mid b_j \Rightarrow \psi_m) : \tau' \end{array}}{L, \delta, \texttt{return}(b_1 \Rightarrow \overline{x_n : \tau_n} \mid \ldots \mid b_j \Rightarrow \overline{x_n : \tau_n}) : \tau \rightharpoondown \psi} \tag{66}$$

Fig. 22 Functional SSA transformation rules—transfer

inserted by our transformation. If x is also F, our transformation sets $x : \tau = y_i$ and emits the appropriate select statements for x. If y is non-flat, the select cannot be eliminated.

```
main = $\lambda_f(x_0)$.
 $b_1$ = $\lambda_b()$.
   $x_1$ = 1.0
   $x_2$ = 2.0
   $x_3$ = 3.0
   $x_4$ = ($x_1$, $x_2$, $x_3$)
   $x_5$ = 4.0
   $x_6$ = 5.0
   $x_7$ = 6.0
   $x_8$ = ($x_5$, $x_6$, $x_7$)
   call(DotProduct, ($x_4$, $x_1$, $x_2$, $x_3$, $x_8$, $x_5$, $x_6$, $x_7$), $b_2$);
 $b_2$ = $\lambda_b$($x_{91}$, $x_{92}$, $x_{93}$, $x_{931}$, $x_{932}$, $x_{933}$).
   $x_{11}$ = 4.5
   $x_{12}$ = 5.5
   $x_{13}$ = 6.5
   $x_{14}$ = ($x_{11}$, $x_{12}$, $x_{13}$)
   call(DotProduct, ($x_{93}$, $x_{931}$, $x_{932}$, $x_{933}$, $x_{14}$, $x_{11}$, $x_{12}$, $x_{13}$));
 goto($b_1$);

DotProduct = $\lambda_f$($X$, $x_1$, $x_2$, $x_3$, $Y$, $y_1$, $y_2$, $y_3$).
 $b_3$ = $\lambda_b()$.
    $z_1$ = Op(*, $x_1$, $y_1$)
    $z_2$ = Op(*, $x_2$, $y_2$)
    $z_3$ = Op(*, $x_3$, $y_3$)
    $a_1$ = Op(+, $z_1$, $z_2$, $z_3$)
    $a_2$ = ($a_1$, $X$, $Y$)
    return($b_2$ => ($a_1$, $X$, $Y$, $y_1$, $y_2$, $y_3$) | halt => $a_2$);
  goto($b_3$);

call(main, unit)
```

Fig. 23 Flattened functional SSA example

The rules for transfer[2] expressions (see Fig. 21) must correctly flatten any arguments based on the formals the transfer's target expects. For instance, if we flatten a function f and produce a flattened function f', a call to f must be translated into a call to f'. Therefore, the arguments the call supplies to f' must be translated based on what the formals of f' expect. Since we constrained our labeling of the dispatch statement to match the labeling of the target block, the arity of the return values and formals will match. The transformation rules select out all nesting levels of all tuples, providing the needed components for all tuples passed out through a transfer.

Applying our transformation and utilizing a labeling that flattens all tuples, we can translate the example program DotProduct as shown in Fig. 23. The application of the transformation $\rightharpoondown$ to DotProduct successfully eliminates all selects. However, the number of parameters passed to the function increases. Since both boxed and unboxed forms of X and Y are utilized in DotProduct, the function takes eight arguments. The specialization of the return to blocks b_2 and halt both blocks to take different representations of the returned tuples. There is some duplication of data because an unboxed representation of the original tuples passed to *DotProduct* is required to be passed to halt.

[2] Because the transformation rules for tail and non-tail calls are identical besides syntactical differences, only the rule for non-tail calls is provided.

Fig. 24 Extended typing domain

$$\tau \in \mathit{Type} ::= \iota \mid (\tau_1 * \ldots * \tau_n) \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 + \tau_2$$

3.3 Correctness

Since functions can return potentially different representations of any returned tuple, our existing type system is not adequate. To support multiple return types, we extend our type system with sum types and add a new type checking rule for transformed programs which replaces the old rule for `return`. We write $\tau_1 + \tau_2$ as the sum of types τ_1 and τ_2 (see Fig. 24). The typechecking rule for `return` is fairly similar to the rule defined for functional SSA, we check if the types of the formals and actuals match for each target block. However, the type returned is the sum of types of the actuals passed to each block.

$$\frac{\begin{array}{c}\Gamma(b_i) = \tau_i \rightarrow \tau_i' \; i = 1, \ldots, j \\ (b_i \Rightarrow (x_{1i} : \tau_{1i}, \ldots, x_{ni} : \tau_{ni}) \; \tau_i = (\tau_{1i} * \cdots * \tau_{ni}) \\ \tau = \tau_l + \cdots + \tau_k \; \tau_l \neq \tau_k \; l, k \in 1, \ldots, j\end{array}}{\Gamma \vdash \texttt{return}(\ldots|b_j \Rightarrow (x_{j1} : \tau_{j1}, \ldots, x_{mj} : \tau_{mj})) : \tau} \tag{67}$$

We now define type safety for the transformed program in Theorem 1.

Theorem 1 *Type safety for a transformed program. Given labeling L, and $P \in$ Prog such that $\Gamma_\perp \vdash P : \tau$ where $L, \delta_\perp, P \rightharpoondown P'$ then $\Gamma_\perp \vdash P' : \tau$.*

We proceed with a proof by contradiction by examining statements and transfers. Assume L, and $P \in$ Prog such that $\Gamma_\perp \vdash P : \tau$ where $L, \delta_\perp, P \rightharpoondown P'$ and $\Gamma_\perp \vdash P' : \tau'$. There are two possible cases, either a statement or a transfer produced a new type.

Transfers—Lemma 1 guarantees that all free variables are bound in the current typing environment. Based on the labeling constraints, we know the labels of the actuals and formals will match. Therefore, based the structure of δ and the transformation rules, the types of the formals and actuals will match.

Statements—Statements that are not transformed remain type-safe. Statements in which the left hand side is labeled F, will have their structure selected out into fresh variables, based on δ. The labeling constraints and structure of δ guarantee the types match between the fresh variables and the selected components of the tuple. The structure of δ preserves the correlation between the fresh variables and tuple components. When selects are transformed, the structure of δ and the labeling constraints guarantee the types match between the removed selected and the fresh variable. We thus arrive at a contradiction since no new types are introduced into the program. The structure of δ preserves the matching between fresh variables and decoupled tuples elements, and labeling constraints force the match between formals and actuals for all transfers, thus τ must equal τ'.

Theorem 2 *Correctness of a transformed program. Given labeling L, and $P \in$ Prog such that, $\rho_\perp \vdash P : \tau \rightarrow_E v$ where $L, \delta_\perp, P \rightharpoondown P'$ then $\rho_\perp \vdash P' : \tau \rightarrow_E v$.*

The definition is standard: if a program evaluates to a value v, we expect its transformed counterpart to also evaluate to v. Since our transformation introduces only new temporary variables, all variables present in program P will occur in the transformed program P' and will have the same runtime value (namely the same binding in ρ). Notice most statements are not transformed; instead new statements are added to the program. We proceed by constructing a proof by contradiction. Assume $\rho_\perp \vdash P : \tau \rightarrow_E v$ where $L, \delta_\perp, P \rightharpoondown P'$ and $\rho_\perp \vdash P' : \tau \rightarrow_E v'$. There are two possible cases: either a transformed statement or transfer generated a different value. We examine each case.

Statements—The only statements that are transformed are select statements of the form $x : \tau = \#i\ y$ which are changed to assignments $(x : \tau = y_i)$ by rules 59 and 60. The variable y_i is extracted from the ith component of y bound in the tuple environment. By the structure of δ, we know y must have been labeled *flat*. By Lemma 2, y's definition occurs prior in the program, thus y is bound in ρ. By rule 55 and Lemma 2, all components of y are also bound in ρ, through application of rules 45 and 46. Namely, the ith component of $\delta(y)$, y_i, is bound to the ith component of y resulting in x being bound to the same value in P as well as P'. The proof for all other statements holds trivially by Lemma 2.

Transfers—By Lemma 2, all free variables in each transfer are bound in ρ. Based on our labeling constraints which specify the matching of labels between actuals and formals, thus guaranteeing appropriate bindings in the tuple environment for each tuple argument, we are able to flatten any argument of the transfer. Therefore, by Lemma 2 and the transformation rules for transfer the value of the formals and actuals match. We thus arrive at a contradiction since no new values are introduced into the program, only decoupled tuples elements, and formals and actuals match on all transfers, v must equal v'.

4 Experimental results

4.1 Labelings

Labelings provide the framework for the flattening algorithm; each labeling corresponding to a flattening strategy. Benchmarks results directly depend on a flattening strategy because a labeling determines the number of flattened tuples constructed for a program. The average data size, length, and nesting level of tuples labeled *flat* also affects program performance. Since a variety of dynamic statistics based on a labeling determine efficiency, it is difficult to ascertain a unique optimal labeling strategy for all programs.

To illustrate the optimization potential and efficiency of tuple flattening, we compare and contrast three strategies: (1) a *bounded* flattening method, (2) an *argument-only* strategy, and (3) a *flatten-all* approach. These three strategies give a good overview of the effects of tuple flattening. Results comparing the three strategies are presented in Sect. 4.2.

As a baseline reference strategy, we utilize *argument-only*. In ML all functions take one argument, a tuple of the arguments passed to that function. The *argument-only* strategy unboxes this tuple, thereby allowing functions to have more than one argument. This is a basic compiler optimization for ML, which leaves all explicitly defined tuples boxed. The corresponding labeling strategy marks all variables in functional SSA as *non-flat*.

The *bounded* flattening methodology flattens tuples only if they will be used mostly in a flat context (determined by heuristics). Locally, we flatten tuples up to three nesting levels if they are the target of select statements. However, for tuples passed between functions, the *bounded* scheme flattens only those tuples created explicitly in bindings. These are user-defined tuples and often the targets of selects. The corresponding labeling strategy labels all variables of tuple type local within blocks and variables of tuple type that are explicitly created as *flat*. All other variables are labeled as *non-flat*.

The *flatten-all* approach is the converse of the *argument-only* strategy. All tuples will be aggressively unboxed to the extent their use-sites will allow. At every definition of a tuple, the unboxed version is generated. Useless code elimination removes any representations which are not used. The only boxed tuples that will remain in the program are tuples supplied as arguments to primitives. The result of this strategy is a program that contains as few boxed tuples as possible. The corresponding labeling strategy marks all tuple variables as *flat*. All other variables as marked *non-flat*.

4.2 Results

The results presented illustrate general trends seen across the MLton benchmark suite[3] and were obtained from running the MLton compiler on an Intel p4 2.4 Ghz computer with 1 gigabyte of memory running Gentoo Linux. The MLton benchmarking script runs and compares various versions of the compiler for each benchmark. Benchmarks are executed ten times and the average of each measurement is reported.

The first graph (see Fig. 25) compares the runtime ratios of both the *flatten-all* and the *bounded* flattener normalized against the *argument-only* strategy. The first, light gray column represents the *flatten-all* methodology and the second, dark gray column the *bounded* flattener. We observe significant speed ups in almost all cases using the *bounded* and the *flatten-all* method when compared to *argument-only*. The cases that showed a degradation of performance (i.e., a run-time ratio greater than 1) for *flatten-all*, suffered from incurred flattening overheads which included data duplication, increased register pressure, and large stack growth. Since the MLton compiler unboxes floating point numbers, we did not unbox any tuple whose constituent components were of type real to limit stack growth. As a result, floating point benchmarks, such as `barnes-hut`, did not benefit from the flattening pass.

In the floating point benchmarks most tuples were not flattened due to the heuristic restriction limiting flattening to non-floating point tuples. The larger amount of non-flat tuples within a program, the greater the data duplication since useless code elimination cannot remove one of the two representations. We also noticed a larger than expected performance gain in programs that contained little to no user-defined tuples, such as `Psdes-rand`. In these test cases, the flattening algorithm unboxed intermediate tuples introduced into the program via other optimization passes and calling conventions. In such cases, almost all tuples were eliminated by our transformation. In programs that contain few user-defined tuples, the majority of the benefit or our optimization results from the elimination of compiler introduced tuples. In general, both the *bounded* and the *flatten-all* strategies performed very well on both higher order and regular benchmarks compared to *argument-only*.

Static program size was not significantly affected by our flattening algorithm. In fact, in many cases program size shrunk after the application of the source level transformation. The graph shown in Fig. 26 depicts a selection of benchmarks and their growth in size for each flattening strategy. Some programs, like `MD5`, were significantly reduced in size. This large reduction is not an anomaly, and occurs because redundant selects and exposed useless variables are eliminated during the third phase of our flattening transformation. Whenever a tuple is flattened, useless code elimination is able to remove components of the tuple which are not utilized by the program. Across forty benchmarks, the average size of the program shrunk by five percent when compared to a *argument-only* strategy. The difference between the *bounded* and the *flatten-all* scheme was negligible. Compile time ratio differences between the *bounded* and the *flatten-all* implementation were negligible and are omitted.

[3] The MLton benchmark suite contains about forty benchmarks with a selection of floating point, high order, and regular benchmarks (see http://www.mlton.org).

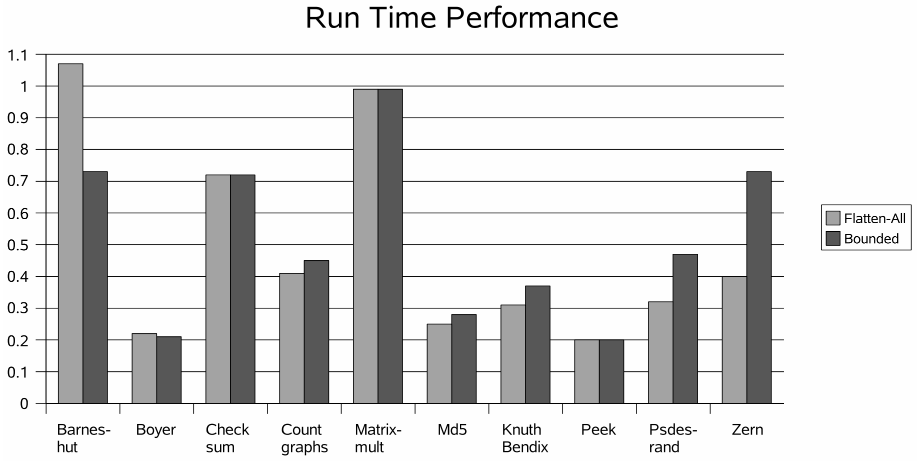

Fig. 25 Run time ratios—normalized vs argument-only

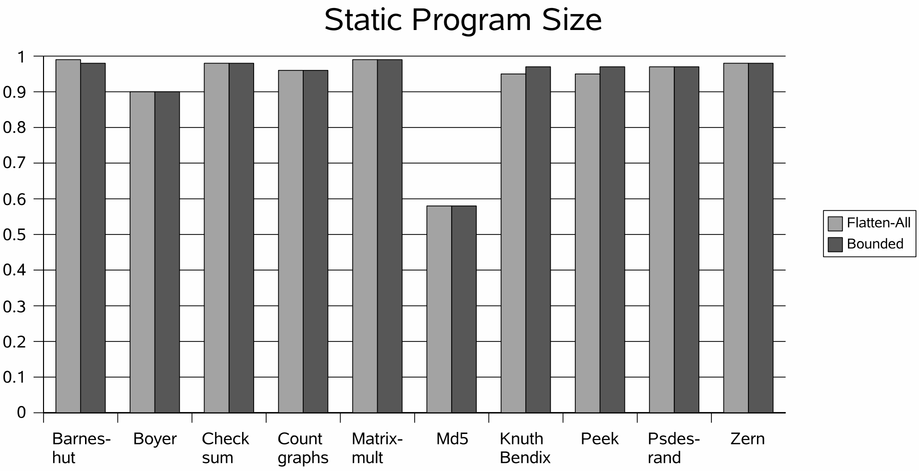

Fig. 26 Program size ratios—normalized vs argument-only

Total allocation in bytes varied greatly between the *flatten-all* and *argument-only* schemes. The Fig. 27 shows total allocation ratios of *flatten-all* and *bounded* normalized vs *argument-only*. In some cases, `MD5` and `Checksum`, total allocation was reduced by a ratio of one hundred. Decreases in total allocation occur because function arguments are passed in registers, useless tuples are eliminated, and uncoupled, useless tuple elements are eliminated through useless code elimination. Increases in total allocation can occur when both boxed and unboxed representations are heavily utilized within a program, resulting in a duplication of data because both representations are stored. In general, *flatten-all* experienced a decrease in total allocation. Increases were rare and usually only occurred in programs utilizing floating point numbers or in cases where useless code elimination was unable to remove one of the tuple's representations. Besides floating point benchmarks, in which case *flatten-all* had a higher allocation, *flatten-all* was similar to *bounded*.

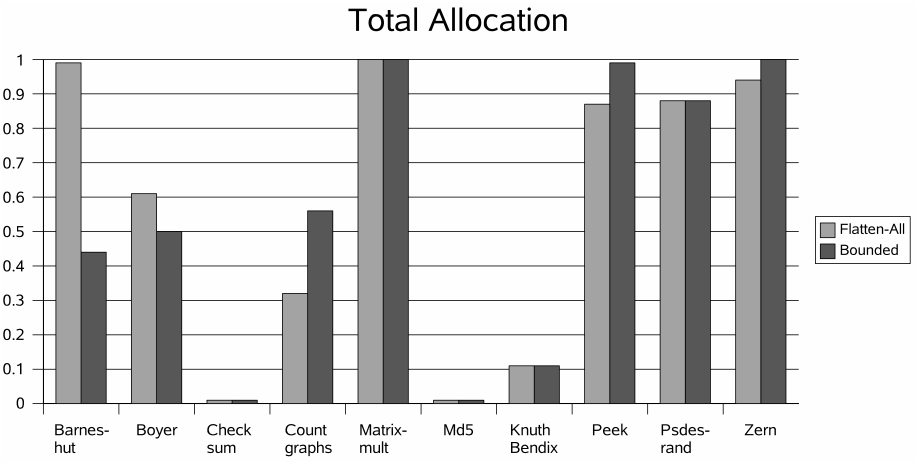

Fig. 27 Total allocation ratios—normalized vs argument-only

Fig. 28 Psdes-rand: pre-flattening

```
L_46 (x_114)
  x_117 = #2 x_114
  x_116 = #1 x_114
  Array_update (x_75, x_76, x_116)
  x_115 = Int32_add (x_76, global_1)
  loop_10 (x_117, x_115)
L_42 (x_118)
  x_120 = #2 x_118
  x_119 = Int32_add (x_72, global_1)
  loop_9 (x_120, x_119)
L_40 (x_121)
  x_124 = #2 x_121
  x_123 = #1 x_121
  Array_update (x_69, x_70, x_123)
  x_122 = Int32_add (x_70, global_1)
  loop_8 (x_124, x_122)
L_36 (x_125)
  x_127 = #2 x_125
  x_126 = Int32_add (x_66, global_1)
  loop_7 (x_127, x_126)
```

4.3 Discussion

Consider the benchmark Psdes-rand presented in Fig. 31. Figures 28 and 29 are functional SSA excerpts from the main loop in Psdes-rand depicting the program before and after the transformation respectively. As the two figures show, all selects are eliminated from the program by our transformation. The selects that are eliminated, were passed through the main loops of Psdes-rand as arguments. By prefetching elements of tuples, we are able to hoist these selects thereby avoiding their repetition within the bodies of the loops. Useless code elimination removes the now unneeded boxed representations, allowing us to transform blocks, such as L_46 and L_42 in Fig. 28, into corresponding blocks, L_46 and L_88 in Fig. 29. The blocks L_46 and L_88 take no heap allocated arguments. This greatly reduces the total allocation of the program, since all tuples are eliminated completely and

Fig. 29 Psdes-rand: post-flattening

```
L_46 (x_146, x_145)
  Array_update (x_75, x_76, x_146)
  x_115 = Int32_add (x_76, global_1)
  loop_10 (x_145, x_115)
L_88 (x_148, x_147)
  x_119 = Int32_add (x_72, global_1)
  loop_9 (x_147, x_119)
L_40 (x_150, x_149)
  Array_update (x_69, x_70, x_150)
  x_122 = Int32_add (x_70, global_1)
  loop_8 (x_149, x_122)
L_87 (x_152, x_151)
  x_126 = Int32_add (x_66, global_1)
  loop_7 (x_151, x_126)
```

$$
\begin{array}{rl}
\pi \in & \mathit{Term} = \mathit{Prog} \mid \mathit{Fun} \mid \mathit{Block} \mid \mathit{Statement} \mid \mathit{Exp} \mid \mathit{Transfer} \mid \mathit{Var} \\
\phi_e \in & \mathit{EvalTerm} = \mathit{Prog} \mid \mathit{Exp} \\
\rho \in & \mathit{Env} = \mathit{Var} \rightarrow \mathit{Value} \\
\Gamma \in & \mathit{TypeEnv} = \mathit{Name} \rightarrow \mathit{Type} \\
v \in & \mathit{Value} = 0 \mid 1 \mid \ldots \mid \mathit{true} \mid \mathit{false} \mid (v_1, \ldots, v_n) \mid \texttt{unit} \\
op \in & \mathit{PrimOp} = + \mid - \mid * \mid \texttt{ref} \mid \texttt{deref} \mid \ldots \\
Bs \in & \mathit{Stack} = (\mathit{Name} \times \mathit{Env})^* \\
\rightarrow_E \in & \mathit{Eval} = \mathit{Prog} \times \mathit{Env} \times \mathit{EvalTerm} \rightarrow \mathit{Value} \\
\rightarrow_T \in & \mathit{EvalT} = \mathit{Env} \times \mathit{Stack} \times \mathit{Term} \rightarrow \mathit{Value} \\
\rightarrow_B \in & \mathit{Bind} = \mathit{Env} \times \mathit{Statements} \rightarrow \mathit{Env} \\
\aleph \in & \mathit{PrimApp} = \mathit{Op} \times \mathit{Values} \rightarrow \mathit{Value} \\
L \in & \mathit{Labeling} = \mathit{Var} \rightarrow \mathit{Label} \\
\delta \in & \mathit{TupleEnv} = \mathit{Labeling} \times \mathit{Var} \rightarrow \mathit{Labeling} \times \mathit{TypedVars} \\
\rightarrow_\delta \in & \mathit{Fold} = \mathit{Labeling} \times \mathit{TupleEnv} \times \mathit{Terms} \\
 & \rightarrow \mathit{Labeling} \times \mathit{TupleEnv} \\
\rightharpoonup_\delta \in & \mathit{Fold} = \mathit{Labeling} \times \mathit{TupleEnv} \times \mathit{Terms} \\
 & \rightarrow \mathit{Labeling} \times \mathit{TupleEnv} \\
\psi \in & \mathit{Terms} \\
\rightharpoondown \in & \mathit{Transform} = \mathit{Labeling} \times \mathit{TupleEnv} \times \mathit{Term} \rightarrow \mathit{Terms} \\
\rightharpoonup_F \in & \mathit{Transform} = \mathit{Labeling} \times \mathit{TupleEnv} \times \mathit{Term} \rightarrow \mathit{Terms} \\
\rightsquigarrow_F \in & \mathit{FoldSel} = \mathit{TupleEnv} \times \mathit{Terms} \rightarrow \mathit{Terms} \\
\rightsquigarrow_S \in & \mathit{Select} = \mathit{TupleEnv} \times \mathit{Term} \rightarrow \mathit{Terms}
\end{array}
$$

Fig. 30 Complete domain listing

their utilized elements are free to be passed in registers. Since data duplication was not a problem, stack growth was bounded by the size of the flattened tuple.

It is clear that *flatten-all* is a large win over *argument-only*. However, when compared to the *bounded* flattening strategy, most benchmarks were equivalent. Some performed better and most floating point benchmarks performed worse. If floating points were left boxed, we suspect that the differences between *flatten-all* and *bounded* on floating point benchmarks would be negligible. We surmise that more sophisticated labeling strategies and labeling schemes will perform better than *flatten-all*. For instance, instead of completely unboxing tuples, we could envision a labeling strategy that labels all tuple components. This would allow certain components to remain boxed by introducing a singleton tuple. Another option is to add more label types so that more information can be stored about a particular tuple and its uses.

In a whole-program compilation environment tuple unboxing is able to avoid many of the problems cited by Leroy. Since our approach is completely coercion free, no costs accrue from changing representations since both are always available where needed. Compared to

```
fun once () =
   let
   fun natFold (start, stop, ac, f) =
      let
 fun loop (i, ac) =
    if i = stop then ac
    else loop (i + 1, f (i, ac))
      in loop (start, ac)
      end
   val niter: int = 4
   open Word32
   fun make (l: word list) =
      let val a = Array.fromList l
      in fn i => Array.sub (a, i)
      end
   val c1 = make [0wxbaa96887, 0wx1e17d32c, 0wx03bdcd3c, 0wx0f33d1b2]
   val c2 = make [0wx4b0f3b58, 0wxe874f0c3, 0wx6955c5a6, 0wx55a7ca46]
   val half: Word.word = 0w16
   fun reverse w = orb (>> (w, half), << (w, half))
   fun psdes (lword: word, irword: word): word * word =
      natFold
      (0, niter, (lword, irword), fn (i, (lword, irword)) =>
       let
  val ia = xorb (irword, c1 i)
  val itmpl = andb (ia, 0wxffff)
  val itmph = >> (ia, half)
  val ib = itmpl * itmpl + notb (itmph * itmph)
       in (irword, xorb (lword, itmpl * itmph + xorb (c2 i, reverse ib)))
       end)
   val zero: word = 0wx13
   val lword: word ref = ref 0w13
   val irword: word ref = ref 0w14
   val needTo = ref true
   fun word () =
      if !needTo
 then
    let
       val (l, i) = psdes (!lword, !irword)
       val _ = lword := l
       val _ = irword := i
       val _ = needTo := false
    in l
    end
      else (needTo := true; !irword)
   fun loop (i, w) =
      if i = 0
 then
    if w = 0wx132B1B67 then ()
    else raise Fail "bug"
      else loop (Int.- (i, 1), w + word())
   in loop (150000000, 0w0)
   end
structure Main =
   struct
      fun doit n =
 if n = 0 then ()
 else (once ()
       ; doit (n - 1))
   end
val _ = Main.doit 2
```

Fig. 31 Psdes-rand: MLton benchmark

runtime type inspection, no typing information is necessary at run time to perform our unboxing technique. All unboxing is handled statically by the transformation. In contrast, our transformation may cause increased register pressure, larger stack frames, and duplication of data. These costs are not unique to our transformation, as other transformations also retain multiple representations.

Although a function may never require the boxed version of a tuple, the overhead of passing a large number of arguments can offset the removal of any memory indirections select statements introduce, especially if the elements of the tuple are large. Nested tuples can also greatly increase stack size, since a tuple can have an arbitrary structure. A tuple containing a complex nesting structure, if flattened completely, may cause exponential stack growth. This is especially true if both representations at each nesting level of the tuple are not eliminated by the useless code elimination pass. Although this occurs infrequently, it is nonetheless an unfortunate side-effect of passing two representations. Passing duplicate data can result in large stack frames since both pointers to heap allocated objects and their unboxed versions are passed. Large stack frames not only are slow to construct, but contribute to a higher rate of register spills.

References, arrays, and potentially other primitive operations may cause an implementation to lose track of tuples. If we store a tuple in a reference cell (or any mutable object), we can no longer guarantee its mapping in the tuple environment. Our transformation is conservative in such cases, treating tuples extracted from mutable datastructures as new tuples. Although SSA handles all potential naming problems, duplicate work is done if this tuple is known. In the presence of must alias information [5], these duplications can be removed.

5 Conclusion

We provided a definition of a tuple flattening optimization which can be incorporated into functional SSA-based compilers. Our transformation rules provide a framework in which different flattening mechanisms (labelings) can be studied and compared formally. Our optimization is insensitive to the depth of tuple nestings. The optimization potential for tuple flattening is significant; for certain benchmarks, improvements range up to 90% over optimized counterparts that do not exploit flattening transformations. Our results indicate that flattening is a reasonable and efficient optimization that can benefit any language that utilizes tuple structures.

Acknowledgements We would like to thank our anonymous reviewers for their comments and suggestions. This work is supported by the National Science Foundation under grants CCF-0701832 and CCR-0509387.

References

1. Cejtin, H., Jagannathan, S., Weeks, S.: Flow-directed closure conversion for typed languages. In: ESOP ’00: Proceedings of the 9th European Symposium on Programming Languages and Systems, London, UK, pp. 56–71 (2000)
2. Davis, M.K.: Deforestation: Transformation of functional program to eliminate intermediate trees (1987)
3. Elsman, M.: Program modules, separate compilation, and intermodule optimization (1999)
4. Hall, C.V., Peyton Jones, S.L., Sansom, P.M.: Unboxing using specialisation. In: GLA (1994)
5. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: Single and loving it: must-alias analysis for higher-order languages. In: POPL ’98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, pp. 329–341 (1998)
6. Kelsey, R.A.: A correspondence between continuation passing style and static single assignment form. SIGPLAN Not. **30**(3), 13–22 (1993)

7. Leroy, X.: Unboxed objects and polymorphic typing. In: POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, pp. 177–188 (1992)
8. Leroy, X.: The effectiveness of type-based unboxing (1997)
9. Minamide, Y., Garrigue, J.: On the runtime complexity of type-directed unboxing. In: ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, pp. 1–12 (1998)
10. Mogensen, T.A.: Glossary for partial evaluation and related topics. High.-Order Symb. Comput. **13**(4), 355–368 (2000)
11. Morrissett, G.: Compiling with types (1995)
12. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of 25th ACM National Conference, Boston, Massachusetts, pp. 717–740. Reprinted in High.-Order Symb. Comput. **11**(4), 363–397 (1998)
13. Shao, Z.: Compiling standard ML for efficient execution on modern machines (1994)
14. Shao, Z., Appel, A.W.: A type-based compiler for standard ML. In: PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, New York, NY, USA, pp. 116–129 (1995)
15. Shao, Z., Appel, A.W.: Efficient and safe-for-space closure conversion. ACM Trans. Program. Lang. Syst. **22**(1), 129–161 (2000)
16. Shivers, O.: Control flow analysis in scheme. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, New York, NY, USA, pp. 164–174 (1988)
17. Shivers, O.: Control-flow analysis of higher-order languages or taming lambda. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. Technical Report CMU-CS-91-145 (1991)
18. Steckler, P.A., Wand, M.: Lightweight closure conversion. ACM Trans. Program. Lang. Syst. **19**(1), 48–86 (1997)
19. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., Lee, P.: TIL: a type-directed, optimizing compiler for ML. SIGPLAN Not. **39**(4), 554–567 (2004)
20. Tofte, M.: A brief introduction to regions. In: ISMM '98: Proceedings of the 1st International Symposium on Memory Management, New York, NY, USA, pp. 186–195 (1998)
21. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N.: A retrospective on region-based memory management. High.-Order Symb. Comput. **17**(3), 245–265 (2004)
22. Tolmach, A., Oliva, D.P.: From ML to Ada: Strongly-typed language interoperability via source translation. J. Funct. Program. **8**(4), 367–412 (1998)
23. Wadler, P.: Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, New York, NY, USA, pp. 45–52 (1984)
24. Wadler, P.: Listlessness is better than laziness II: composing listless functions In: On Programs as Data Objects, New York, NY, USA, pp. 282–305 (1985)
25. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (1989). Special issue on ESOP'88, the Second European Symposium on Programming, Nancy, France, March 21–24, 1988
26. Wand, M., Siveroni, I.: Constraint systems for useless variable elimination. In: POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, pp. 291–302 (1999)