Yale University
Department of Computer Science

Subtransitive CFA using Types

Bratin Saha
Yale University
New Haven, CT

Nevin Heintze
Bell Laboratories
Murray Hill, NJ

Dino Oliva
Bell Laboratories
Murray Hill, NJ

YALEU/DCS/TR-1166
October 9, 1998

The first author was sponsored in part by a summer internship at Bell Laboratories and in part by the DARPA ITO under the title "Software Evolution using HOT Language Technology", DARPA Order No. D888, issued under Contract No. F30602-96-2-0232. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Subtransitive CFA using Types

Bratin Saha
Yale University

Nevin Heintze
Bell Laboratories

Dino Oliva
Bell Laboratories

October 9, 1998

Abstract

We present an experimental evaluation of Heintze and McAllester's linear-time subtransitive CFA algorithm, in the context of SML/NJ v110.7. As described in [9], linear-time termination of the algorithm depends on the existence of a simple typing of the program such that the type tree at each node has a bounded size. We show that this condition is violated by many programs, especially heavily functorized ones, and so the type-directed version of the algorithm (which explores a program's entire type structure) is only practical on small programs. We also show that the demand-driven variant of the algorithm does not always terminate for programs with polymorphic type. Our main result is that a hybrid algorithm that combines both approaches avoids both the problems. Whereas Heintze and McAllester's original formulation relied on the *existence* of types to bound the runtime of the algorithm, our work shows that an effective implementation must actually *use* the types to control termination.

1 Introduction

Some form of control-flow analysis (CFA) is a standard component of optimizing compilers when the flow-of-control is not explicit in the program text (such as programs written in functional and object-oriented languages). The purpose of control-flow analysis is to determine an approximation of the program's control-flow by computing an upper bound on the functions/methods that can be called/invoked from each call/method-invocation site in the program.

In compilers for functional languages, this analysis has largely been just a simple syntactic reasoning to determine which functions are called once, which functions are "known" functions and which functions are "escaping". Such an analysis is simple and cheap, but effective for mostly first-order programs. However for programs that make significant use of higher-order features, this analysis is very crude, and may result in many lost opportunities for optimization.

Over the last 20 years, a number of more aggressive approaches to control-flow analysis have been considered. Starting from the early work by Jones [11, 12] and then Shivers [15], there has been an explosion of interest in issues such as extending the basic analysis to data structures, references, arrays, implementation strategies and polyvariance, e.g. [2, 6, 1, 5, 10, 16, 4].

The first author was sponsored in part by a summer internship at Bell Laboratories and in part by the DARPA ITO under the title "Software Evolution using HOT Language Technology", DARPA Order No. D888, issued under Contract No. F30602-96-2-0232. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Two basic algorithms have emerged. The first, often called 0CFA, is based on "flowing" values in a way that mimics program execution. When formulated as constraints, it uses inclusions. This algorithm has cubic time complexity (in the size of the input program) and recent work shows that, in the general case, we cannot hope to do better [8]. The other algorithm is based on unification, and involves a bi-directional flow of information – in essence, if one node can "flow" to another, then the two nodes are unified. When formulated as constraints, it uses equations. The advantage of this algorithm is that it runs in linear-time (in the size of the program); its disadvantage is that the unification steps can propagate information "backwards", which can lead to less accurate analysis results.

Much recent work has focused on understanding and reconciling the differences between these two approaches. In practice, it has been observed that analyses based on 0CFA rarely exhibit cubic behavior, but they often behave non-linearly, and this has discouraged their wider use in production compilers. Some papers have addressed methods for improving the scalability of 0CFA-style analyses by removing redundancies [5] and by recognizing "equivalent" nodes and collapsing them [4] (strictly speaking, [4] addresses points-to-analysis for C, but the techniques are directly applicable to 0CFA). Other papers have dealt with making unification-based analyses more accurate. For example, [3] shows that by changing the way calls and returns are matched, one can improve the accuracy of the unification-based approach, so that for some benchmarks, the results are essentially as accurate as those obtained by using the cubic time algorithm. In yet another approach, [9] develop a linear-time algorithm that computes the same information as the 0CFA for programs with bounded-type size. (Strictly speaking, their algorithm computes a graph whose transitive closure is identical to the results of 0CFA; they argue that many applications of CFA can be adapted to directly use this "subtransitive control-flow graph" in linear-time.) In this paper, we present an experimental evaluation of the linear-time subtransitive CFA algorithm (LT-ST-CFA), in the context of SML/NJ v110.7.

As described in [9], linear-time termination of the algorithm depends on the existence of a simple typing of the program such that the type tree at each node has a bounded size. A key question raised in [9] is: do programs have bounded type size in practice and how big is the bound? Early results in [9] for some small benchmarks (up to about 1000 lines of ML), and substantial folklore evidence suggest that most programs people write have bounded types and that the bound is quite small. However, very recent work [14] suggests otherwise. In section 3, we present a detailed type-size study that includes more benchmarks and measures type size in a way that is directly applicable to the LT-ST-CFA algorithm. Although our results are not as negative as the ones in [14], they provide strong evidence that a naive type-based approach to the LT-ST-CFA algorithm is not tenable. More concretely, the LT-ST-CFA algorithm consists of two phases: the first traverses the program and builds a graph; the second phase applies a collection of closure rules to this graph and performs the actual CFA. If the types of a program are used as templates to guide the application of the closure rules, (in phase two of the algorithm), then our results show that the approach is not scalable.

[9] suggest that instead of using types to guide the closure rules, we can make the algorithm demand-driven. The idea here is that in the second phase of the algorithm, we apply the closure rules only when (and if) they are needed. The intuition is that in many cases we need not explore the entire type tree for each node in the program. However, as we show in Section 4, the extension to polymorphically typed programs (described in [9]) interacts badly with the demand-driven approach. Specifically, the demand-driven approach does not always terminate on polymorphically typed bounded-type programs.

Our main result is that a hybrid algorithm that is demand driven but uses type information to provide an explicit bound on the application of the closure rules is fast and effective. The algorithm is polyvariant, and focuses on control-flow of functions (e.g. it provides a correct but trivial treatment of datatypes). Whereas Heintze and McAllester's original formulation relied on the *existence* of types to bound the runtime of the algorithm, our work shows that effective implementations of this algorithm must actually *use* the types to control termination.

2 Control-Flow Analysis and the LT-ST-CFA Algorithm

The purpose of control-flow analysis is to determine the set of functions that can be called from different call sites. We formalize this by using a lambda calculus in which each abstraction is labelled.

$$\mathrm{e} \quad ::= \quad x \mid \lambda^l x.e \mid @e_1 e_2$$

Such labels allow us to trace a program's control flow. The purpose of control-flow analysis is to associate a set of labels $L(e)$ with each sub-expression e of the program such that if e reduces to an abstraction labeled l during execution of the program, then $l \in L(e)$. In other words, control-flow analysis gives a conservative approximation of the abstractions that can be encountered at each expression during execution of a program. We can now define standard control-flow analysis as the least such mapping of labels satisfying the following conditions:

- for an abstraction $\lambda^l x.e$, we have that $l \in L(\lambda^l x.e)$
- for an application $@ee_2$, if $l_1 \in L(e)$ and l_1 labels the abstraction $\lambda^{l_1} x.e_1$, then
  - $L(x) \supseteq L(e_2)$ in the body of the abstraction labelled by l_1
  - $L(@ee_2) \supseteq L(e_1)$

The standard control-flow analysis algorithm essentially computes the least fixed point; it initialises all the label sets to $\emptyset$, and continues adjusting the label sets until the above conditions are satisfied for all the subexpressions in the program.

This algorithm can be reformulated as a graph algorithm. Nodes in the graph represent program sub-expressions; edges are added according to the rules in Figure 1.

$$(ABS) \qquad \frac{}{\lambda^l x.e \rightarrow \lambda^l x.e}$$

$$(APP\text{-}1) \qquad \frac{e_1 \rightarrow \lambda^l x.e \qquad @e_1 e_2 \in P}{x \rightarrow e_2}$$

$$(APP\text{-}2) \qquad \frac{e_1 \rightarrow \lambda^l x.e \qquad @e_1 e_2 \in P}{@e_1 e_2 \rightarrow e}$$

$$(TRANS) \qquad \frac{e_1 \rightarrow e_2 \qquad e_2 \rightarrow e_3}{e_1 \rightarrow e_3}$$

Figure 1: Edge Formation Rules

An edge $e_1 \rightarrow e_2$ implies that all nodes reachable from e_2 are also reachable from e_1. Given this transition system, standard control-flow analysis may now be reformulated as follows – given a program expression e, find all abstractions $\lambda^{l'} x.e'$ such that $e \rightarrow \lambda^{l'} x.e'$ is derivable from the transition rules. This reformulation clearly shows the interdepedence of the transitive closure and the edge addition components of the algorithm: in fact, the algorithm is an instance of dynamic transitive closure.

The key idea behind the Heintze/McAllester algorithm is to reformulate the rules to decouple transitive closure from edge addition. For completeness, we extend the language introduced earlier in this section to include labelled records $(e_1, \ldots, e_p)$ and projections $\pi_i e$.

$$(ABS\text{-}1) \qquad \frac{\lambda^l x.e \in P}{x \rightarrow Dom(\lambda^l x.e)}$$

$$(ABS\text{-}2) \qquad \frac{\lambda^l x.e \in P}{Ran(\lambda^l x.e) \rightarrow e}$$

$$(APP\text{-}1) \qquad \frac{@e_1 e_2 \in P}{Dom(e_1) \rightarrow e_2}$$

$$(APP\text{-}2) \qquad \frac{@e_1 e_2 \in P}{@e_1 e_2 \rightarrow Ran(e_1)}$$

$$(RECORD) \qquad \frac{(e_1, \ldots, e_p) \in P}{Proj_i((e_1, \ldots, e_p)) \rightarrow e_i}$$

$$(SELECT) \qquad \frac{\pi_i e \in P}{\pi_i e \rightarrow Proj_i(e)}$$

$$(CLOSE\text{-}DOM) \qquad \frac{n_1 \rightarrow n_2}{Dom(n_2) \rightarrow Dom(n_1)}$$

$$(CLOSE\text{-}RAN) \qquad \frac{n_1 \rightarrow n_2}{Ran(n_1) \rightarrow Ran(n_2)}$$

$$(CLOSE\text{-}PRJ) \qquad \frac{n_1 \rightarrow n_2}{Proj_i(n_1) \rightarrow Proj_i(n_2)}$$

Figure 2: The Edge Formation Rules

In the graph algorithm above, nodes were program expressions. For the LT-ST-CFA algorithm, we extend nodes as follows:

$$n \quad ::= \quad e \mid Dom(n) \mid Ran(n) \mid Proj_i(n)$$

Nodes representing program expressions are called *basic nodes*; nodes of the form $Dom(n)$, $Ran(n)$ and $Proj_i(n)$ are called *administrative* nodes. The administrative nodes are the glue that track the flow of values through a program. If node n represents a function, then the node $Dom(n)$ represents the domain of the node n; i.e. the set of values that can flow to the function's argument. Conversely $Ran(n)$ represents the range of the node n; the set of values that the function can return. If n represents a record, then $Proj_i(n)$ represents the set of values that may flow into the i^{th} component of the record. The LT-ST-CFA algorithm can now be formulated as in Figure 2. An edge $e_1 \rightarrow e_2$ implies that the set of values that e_1 can reduce to is a superset of the set of values that e_2 can reduce to.

The main issue with this algorithm is that the application of the rules CLOSE-DOM, CLOSE-RAN and CLOSE-PRJ is open ended. One way to do this is to use the program types to guide these rules. That is, we apply the closure rules so that all nodes compatible with the type of a basic node are generated. For example if e has type $\text{int} \rightarrow \text{int} \rightarrow \text{int}$, then we only need to build the nodes $Dom(e)$, $Ran(e)$, $Dom(Ran(e))$ and

Bmark	Avg. Type Size	Type Size Range					No. Of Types
		$> \mathbf{10}$	$> 10^2$	$> 10^3$	$> 10^4$	$> 10^5$	
BHut	21	219	64	8	-	-	2140
Ray	27	233	83	2	-	-	947
Yacc	36	3692	510	56	-	-	14174
Vliw	73	880	874	197	-	-	9995
CML	81	2993	1359	34	27	-	11072
CM	1490	3260	752	242	239	90	15299

Figure 3: Average Type Size – Including Dataypes

$Ran(Ran(e))$. Hence the performance of the algorithm is directly tied to the type tree size of the program nodes. The other approach is to make the algorithm demand-driven. The basic idea here is that additional preconditions are added to the rules so that they fire only if there is an in-edge to the node from which the rule produces an out-edge. For example, in the CLOSE-DOM rule we add the precondition $n' \to Dom(n_2)$ i.e. there is an edge from some random node n' to $Dom(n_2)$. See [9] for further details and correctness arguments.

3 Type Size

Type size is critical to the practicality of the LT-ST-CFA algorithm, particularly for the variant that uses types to control the application of the closure rules. We have carried out detailed type size measurements for a large number of benchmarks under SML/NJ 110.7. Our results are summarized in Figures 3 and 4. The last column gives the total number of program types while the *Type Size* columns show the number of types in a given range. We measure both the total type size (Figure 3) and also the type size excluding datatypes (Figure 4). In this paper, we focus on flow of functions and provide a trivial approximation of datatypes (we use one node for all data constructors, and as a first approximation, any function put in a data structure flows to any site where a function is removed from a data structure). Hence, the relevant type size metric for our purposes is the one that ignores the size of datatypes. It is clear that for a number of programs, expanding the nodes using type-trees is problematic. To give some intuition about what kinds of programs cause problems, consider the following code fragment where type size grows exponentially:

Bmark	Avg. Type Size	Type Size Range					No. Of Types
		$> \mathbf{10}$	$> 10^2$	$> 10^3$	$> 10^4$	$> 10^5$	
BHut	8.82	61	26	0	0	0	1319
Yacc	6.08	346	70	0	0	0	5477
Vliw	2.18	30	3	0	0	0	3728
CML	4.25	128	36	0	0	0	3826
CM	335.34	114	185	225	87	2	6322

Figure 4: Average Type Size – Excluding Datatypes

```
fun pair x = (x,x)

val x2 = pair 1 : int * int

val x4 = pair x2 : (int * int) * (int * int)

val x8 = pair x4 : ((int * int) * (int * int)) * ...

...
```

While code like this does not appear in practice at the core-language level, they do occur frequently at the module level. Functors often expose an input module via multiple access paths and this behavior is analogous to our *pair* function above. Furthermore, this programming style is often required to specify sharing constraints and occurs in large pieces of code like `CM`

4 The Demand-Driven Algorithm and Polymorphism

[9] presents a demand driven variant of the LT-ST-CFA algorithm. Section 5 of [9] presents extensions for bounded-type polymorphic programs (a polymorphically typed program has type size bounded by k if the sum of the type tree sizes of the expressions in the let-expanded version of the program are bounded by $k.n$, where n is the size of the original program). The termination argument for the polymorphic case is only relevant to the variant of the algorithm in which types are used to guide the closure rules. Contrary to what is implicitly stated in that section, the demand-driven algorithm does not always terminate on bounded-type polymorphic programs.

Intuitively, this is because a polymorphically typed program when stripped of its types and type checked monomorphically, may require recursive types in its monomorphic typing. In essence, the demand-driven LT-

ST-CFA algorithm blindly tries to construct a monomorphic type for a program; unfortunately, it may not terminate if this typing happens to require recursive types. To illustrate this, consider the following program:

```
fun id x = x
val _ = id ( (id id) id)
```

This program type checks under the Hindley-Milner polymorphic type discipline; however, all of its monomorphic typings involve recursion. The demand-driven LT-ST-CFA algorithm does not terminate on this example.

5 Our Algorithm

The core structure of our algorithm is based on the demand-driven LT-ST-CFA algorithm. In addition, we use types to control the behaviour of the closure rules; that is, a closure rule is applied if *both* it is requested and also the types say that the request is "valid". Intuitively, we use the set of monotypes resulting from all instantiations of polymorphic functions in a program. In practice, we make use of the fact that a variable cannot be instantiated to itself in the absence of polymorphic recursion. So, when an administrative node is built from an already existing node, we check that this invariant is satisfied.

Our algorithm is implemented in the SML/NJ compiler and is based on the FLINT [14] intermediate language. FLINT (Figure 5) is based upon a predicative variant of the Girard-Reynolds polymorphic λ-calculus [17, 18]. We use $T(\mu)$ to denote the type corresponding to the constructor μ. The terms are an explicitly typed λ-calculus (but in A-normal form) with explicit constructor abstraction ($\Lambda^l t.e$) and application ($x[\mu]^l$). We use $(x_1, \ldots, x_p)^l$ to denote a record and ${\pi^l}_i x$ to denote selection from a record. Even though the calculus is in A-normal form, we attach labels, l, to expressions to identify them. This is useful in presenting the algorithm.

$$\begin{array}{llll} (\mathit{cons}) & \mu & ::= & t \mid \texttt{Int} \mid \mu_1 \to \mu_2 \mid (\mu_1, \times \ldots \times, \mu_p) \\ (\mathit{types}) & \sigma & ::= & T(\mu) \mid \sigma_1 \to \sigma_2 \mid (\sigma_1, \times \ldots \times, \sigma_p) \mid \forall t.\sigma \\ (\mathit{terms}) & e & ::= & i \mid x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid (x_1, \ldots, x_p)^l \mid {\pi^l}_i x \mid \lambda^l x{:}\sigma.e \mid @^l x_1 x_2 \mid \Lambda^l t.e \mid x[\mu]^l \end{array}$$

Figure 5: Syntax of the FLINT calculus

In the algorithm, we do not need to differentiate between the core-language and the module-language. Therefore, in the rest of the paper, we will only refer to expressions with module types; that is, $\sigma_1 \to \sigma_2$ and $(\sigma_1, \times \ldots \times, \sigma_p)$. The same rules will hold for the corresponding core-language constructs as well.

Nodes are represented by the following grammar –

$$\begin{array}{llll} (\mathit{id}) & i & ::= & l \mid \mathit{Dom}(i) \mid \mathit{Ran}(i) \mid \mathit{Proj}_s(i) \\ (\mathit{Nodes}) & n & ::= & \langle i, \sigma \rangle \end{array}$$

A node is now a tuple, $\langle i, \sigma \rangle$, consisting of an identifier (i) and a type (σ). An identifier is either a label, l, or is of the form $\alpha^p l$, where l is a label and $\alpha^k = \mathit{Dom}/\mathit{Ran}/\mathit{Proj}_s$ for $1 <= k <= p$. A node of the form

$\langle l, \sigma \rangle$ is called a base node and represents a program expression. In fact, it represents the program expression labelled l. The type σ is the same as the expression type.

A node (n) of the form $\langle \alpha^k(l), \sigma_k \rangle$ is an administrative node. The rules for deriving the type of administrative nodes are given in Figure 7. The nodes $\langle \alpha^i(l), \sigma_i \rangle$ for $0 <= i < k$ are called the ancestors of the node n $(\mathcal{A}(n))$. A *chain* refers to an ordered sequence of the ancestors $(\langle l, \sigma \rangle \ldots \langle \alpha^k(l), \sigma_k \rangle)$. The node $\langle l, \sigma \rangle$ is called the base node for all the nodes in the chain.

In the rest of the paper, we will assume that the label of a node representing a variable is the variable name itself. We will also sometimes use the node representing an expression to refer to the expression.

The transition system is shown in Figure 6. The algorithm assumes that the type of every sub-expression is known, and moreover, that the input program is well-typed. (Since FLINT is explicitly typed, the type of every sub-expression can be inferred beforehand in a single pass over the code). Edges have the same meaning as before; an edge, $e_1 \rightarrow e_2$, implies that the set of values that e_1 can reduce to is a superset of the set of values that e_2 can reduce to.

The transition system in Figure 6 is written as a series of sequents of the form –

$$\Gamma \vdash e : \sigma \Rightarrow E \ ; \ n$$

Γ is an environment mapping program variables to nodes; e is the expression whose graph is being constructed and has type σ; the transition system adds the edges E to the graph; and n is the node returned by the application of the rule.

Most of the rules in Figure 6 follow in a straightforward way. Consider the (*fn*) rule. It says that the bound variable in an abstraction takes its value from the domain of the abstraction. Moreover, the range of an abstraction is the value returned by its body. Similarly, the $@^l x_1 x_2$ rule says that x_2 is a value for the domain of x_1, and the result of the application is the range of x_1. The (*rec*) rule says that the s^{th} projection of the record takes its value from x_s.

We will formalise a few terms before we present the closure rules. First define the following predicates (t denotes a type variable) –

$$\begin{array}{llllll} \vdash^e isarrow(\sigma) & if & \sigma = \sigma_1 \rightarrow \sigma_2 & \sigma = \forall t.\sigma' & \sigma = t \\ \vdash^e isrec(\sigma) & if & \sigma = (\sigma_1, \times \ldots \times, \sigma_p) & \sigma = t & \end{array}$$

The rules for forming the type of the administrative nodes (in the closure phase) are shown in Figure 7. (t again denotes a type variable.)

We next define a node congruence. When the closure rules are applied, we consider only one node from each congruence class. Two nodes are congruent $(\langle i_1, \sigma_1 \rangle \equiv \langle i_2, \sigma_2 \rangle)$ if –

- Both the nodes have the same base node
- $\sigma_1 = \sigma_2 = t$ (a type variable)

When the type of the newly formed administrative node (in Figure 7) is derived from the parent node (the first four rules), we are essentially traversing the type tree. In the other cases, (when the type of the parent node is a type variable), we are instantiating the type (which is a type variable) of the parent node.

$$(int) \quad \frac{n = \langle l, \mathtt{Int} \rangle \qquad l \textit{ a new label}}{\Gamma \vdash i : \mathtt{Int} \Rightarrow \emptyset \ ; \ n}$$

$$(var) \quad \frac{\Gamma(x) = n}{\Gamma \vdash x : \sigma \Rightarrow \emptyset \ ; \ n}$$

$$(tfn) \quad \frac{\Gamma \vdash e : \sigma \Rightarrow E \ ; \ n' \qquad n = \langle l, \forall t.\sigma \rangle}{\Gamma \vdash \Lambda^l t.e : \forall t.\sigma \Rightarrow E \uplus \langle Ran(l), \sigma \rangle \rightarrow n' \ ; \ n}$$

$$(fn) \quad \frac{n_x = \langle x, \sigma \rangle \qquad \Gamma[x \mapsto n_x] \vdash e : \sigma' \Rightarrow E \ ; \ n' \qquad n = \langle l, \sigma \rightarrow \sigma' \rangle}{\Gamma \vdash \lambda^l x : \sigma . e : \sigma \rightarrow \sigma' \Rightarrow E \uplus n_x \rightarrow \langle Dom(l), \sigma \rangle \uplus \langle Ran(l), \sigma' \rangle \rightarrow n' \ ; \ n}$$

$$(app) \quad \frac{\Gamma(x_1) = \langle i_1, \sigma_2 \rightarrow \sigma_1 \rangle \qquad \Gamma(x_2) = n_2 \qquad n = \langle l, \sigma_1 \rangle}{\Gamma \vdash @^l x_1 x_2 : \sigma_1 \Rightarrow \langle Dom(i_1), \sigma_2 \rangle \rightarrow n_2 \uplus n \rightarrow \langle Ran(i_1), \sigma_1 \rangle \ ; \ n}$$

$$(tapp) \quad \frac{\Gamma(x) = \langle i_1, \forall t.\sigma \rangle \qquad n = \langle l, \sigma[\mu/t] \rangle}{\Gamma \vdash x[\mu]^l : \sigma[\mu/t] \Rightarrow n \rightarrow \langle Ran(i_1), \sigma \rangle \ ; \ n}$$

$$(rec) \quad \frac{\Gamma(x_1) = n_1 \ \ldots \ \Gamma(x_p) = n_p \qquad n = \langle l, (\sigma_1, \times \ldots \times, \sigma_p) \rangle}{\Gamma \vdash (x_1, \ldots, x_p)^l : (\sigma_1, \times \ldots \times, \sigma_p) \Rightarrow \langle Proj_s(l), \sigma_s \rangle \rightarrow n_s \ ; \ n}$$

$$(sel) \quad \frac{\Gamma(x) = \langle i, (\sigma_1, \times \ldots \times, \sigma_p) \rangle \qquad n = \langle l, \sigma_k \rangle}{\Gamma \vdash \pi^l{}_k x : \sigma_k \Rightarrow n \rightarrow \langle Proj_k(i), \sigma_k \rangle \ ; \ n}$$

$$(let) \quad \frac{\Gamma \vdash e_1 : \sigma_1 \Rightarrow E_1 \ ; \ n_1 \qquad n_x = \langle x, \sigma_1 \rangle \qquad \Gamma[x \mapsto n_x] \vdash e_2 : \sigma_2 \Rightarrow E_2 \ ; \ n_2}{\Gamma \vdash \mathtt{let}\ x = e_1 \ \mathtt{in}\ e_2 : \sigma_2 \Rightarrow E_1 \uplus E_2 \uplus n_x \rightarrow n_1 \ ; \ n_2}$$

Figure 6: The Basic Edges

In a language without polymorphic recursion, a type variable can never be instantiated to itself. The node congruence enforces this and is implemented by the *TyVar(_,_)* predicate –

$$\begin{array}{rll} TyVar(n, \sigma) := false & if & \sigma = t \ \ and\ for\ some\ \langle i', \sigma' \rangle \in \mathcal{A}(n),\ \sigma = \sigma' \\ TyVar(n, \sigma) := true & & otherwise \end{array}$$

The closure rules are shown in Figure 8. In all the rules, n is the new node being constructed, while the other nodes already exist in the graph. These rules are applied after the basic edges (Figure 6) have been added.

Consider the (*Dom*) rule. If there is an edge $n_1 \rightarrow n_2$, then while applying n_1, we may actually be applying n_2. Therefore, the domain of n_2 must include the domain of n_1; hence we add the edge $Dom(n_2) \rightarrow Dom(n_1)$. The *isarrow* predicate ensures that the type of n_1 allows the creation of $Dom(n_1)$. ($Dom(n_2)$ already exists in the graph.) The *dom* function returns the type of the *Dom* node (the newly formed node). The $TyVar(\_, \_)$ predicate implements the node congruence. The $n' \rightarrow \langle Dom(i_2), \sigma_2 \rangle$ condition enforces the demand driven nature. (n' is any node that already exists in the graph).

$$
\begin{array}{lcl}
dom(\sigma_1 \rightarrow \sigma_2, \sigma) & := & \sigma_1 \\
ran(\sigma_1 \rightarrow \sigma_2, \sigma) & := & \sigma_2 \\
ran(\forall t.\sigma', \sigma) & := & \sigma' \\
sel_k((\sigma_1, \times \ldots \times, \sigma_p), \sigma) & := & \sigma_k \\
dom(t, \sigma) & := & \sigma \\
ran(t, \sigma) & := & \sigma \\
sel_k(t, \sigma) & := & \sigma
\end{array}
$$

Figure 7: The type formation rules for administrative nodes

$$
(Ran) \quad \frac{\begin{array}{c} n_1 = \langle i_1, \sigma' \rangle \qquad n_2 = \langle i_2, \sigma_2 \rangle \\ isarrow(\sigma_2) \qquad \sigma = ran(\sigma_2, \sigma_1) \\ n_1 \rightarrow n_2 \qquad n = \langle Ran(i_2), \sigma \rangle \qquad TyVar(n, \sigma) \qquad n' \rightarrow \langle Ran(i_1), \sigma_1 \rangle \end{array}}{\langle Ran(i_1), \sigma_1 \rangle \rightarrow n}
$$

$$
(Proj) \quad \frac{\begin{array}{c} n_1 = \langle i_1, \sigma' \rangle \qquad n_2 = \langle i_2, \sigma_2 \rangle \\ isrec(\sigma_2) \qquad \sigma = sel_k(\sigma_2, \sigma_1) \\ n_1 \rightarrow n_2 \qquad n = \langle Proj_k(i_2), \sigma \rangle \qquad TyVar(n, \sigma) \qquad n' \rightarrow \langle Proj_k(i_1), \sigma_1 \rangle \end{array}}{\langle Proj_k(i_1), \sigma_1 \rangle \rightarrow n}
$$

$$
(Dom) \quad \frac{\begin{array}{c} n_1 = \langle i_1, \sigma_1 \rangle \qquad n_2 = \langle i_2, \sigma' \rangle \\ isarrow(\sigma_1) \qquad \sigma = dom(\sigma_1, \sigma_2) \\ n_1 \rightarrow n_2 \qquad n = \langle Dom(i_1), \sigma \rangle \qquad TyVar(n, \sigma) \qquad n' \rightarrow \langle Dom(i_2), \sigma_2 \rangle \end{array}}{\langle Dom(i_2), \sigma_2 \rangle \rightarrow n}
$$

Figure 8: The Closure Rules

Arrays and references are treated similarly; therefore, we will only discuss the case of arrays. The handling of arrays is a little complicated since array update requires a contra-variant closure rule, while array subscription requires a co-variant closure rule. We introduce two additional administrative nodes, the $Set()$ and the $Get()$. During the first phase, the creation of an array with label l and type σ_{arr} leads to the formation of the nodes – $\langle l, \sigma_{arr} \rangle$, $\langle Set(l), \sigma \rangle$, and $\langle Get(l), \sigma \rangle$. We also add the edge, $\langle Get(l), \sigma \rangle \rightarrow \langle Set(l), \sigma \rangle$, between the get and the set nodes. If the array is updated with the value v, then we add the edge $\langle Set(l), \sigma \rangle \rightarrow n_v$, where n_v is the node corresponding to the value v. If the array is subscripted, then we return the *get* node, $\langle Get(l), \sigma \rangle$.

The closure rules for arrays are shown in Figure 9. It is straightforward to define the $isarray$ predicate and the $arrget$ and $arrset$ functions.

We model exceptions by introducing a single exception node that links together all exceptions.

One main advantage of FLINT is that we have direct access to types, which is critical for our algorithm. These types are also used to provide a generic form of node polyvariance that significantly improves the

$$
(Get)\quad \frac{n_1 \to n_2 \qquad \begin{array}{c} n_1 = \langle i_1, \sigma' \rangle \\ isarray(\sigma_2) \\ n = \langle Get(i_2), \sigma \rangle \end{array} \qquad \begin{array}{c} n_2 = \langle i_2, \sigma_2 \rangle \\ \sigma = arrget(\sigma_2, \sigma_1) \\ TyVar(n, \sigma) \end{array} \qquad n' \to \langle Get(i_1), \sigma_1 \rangle}{\langle Get(i_1), \sigma_1 \rangle \to n}
$$

$$
(Set)\quad \frac{n_1 \to n_2 \qquad \begin{array}{c} n_1 = \langle i_1, \sigma_1 \rangle \\ isarray(\sigma_1) \\ n = \langle Set(i_1), \sigma \rangle \end{array} \qquad \begin{array}{c} n_2 = \langle i_2, \sigma' \rangle \\ \sigma = arrset(\sigma_1, \sigma_2) \\ TyVar(n, \sigma) \end{array} \qquad n' \to \langle Set(i_2), \sigma_2 \rangle}{\langle Set(i_2), \sigma_2 \rangle \to n}
$$

Figure 9: The Array Closure Rules

accuracy of the algorithm with little additional cost. One disadvantage of FLINT is that it uses a De Bruijn index representation for types [14]; and as a result, some of the type operations required by our algorithm were not trivial to implement.

Figure 10 shows that this hybrid algorithm is effective in practice, and in particular that it is much more robust than either the demand-driven or type-driven approaches. We use ∞ to indicate benchmarks that did not terminate in two minutes. In the case of the demand-driven benchmarks, we suspect that the analysis was non-terminating. $\star$ indicates that reliable timings are not available yet. All times are in seconds on a Pentium 200MHz, 64MB, 512K secondary cache, running Linux, and represent the least of three runs of each benchmark. *Ratio* gives the ratio of the analysis time to the total compilation time. $Nvars$ is the total number of program variables. $Node_{tot}$ refers to the total number of nodes formed during the analysis and $Time_{tot}$ refers to the total analysis time. *Type Tree* is the total size of the types, (but excluding datatypes), in the program (from Figure 4). Note that while the typesize measurement completely ignores datatypes, the algorithm, however, must generate nodes for datatypes and apply the closure rules to these nodes; so that the analysis is correct. This is the reason that in Figure 10 the number of nodes in the type-driven approach is greater than the size of the type tree.

We remark that our implementation was designed to be very flexible so that we could investigate a number of implementation strategies. Hence, the size of the node counts for each benchmark is the main focus of our results. The timings are included only for completeness; they only represent an upper bound on the running time, but not a representative time for a tuned implementation.

6 Conclusion

We have shown that both the type-driven and the demand-driven versions of the LT-ST-CFA algorithm do not scale well in practice. We have presented a hybrid algorithm that combines both demand-driven and type-driven features and performs well over a variety of benchmarks.

While our current system provides accurate node counts, it has traded flexibility for performance, and therefore does not provide a good basis for evaluating analysis time. We are also porting other analyses to the FLINT intermediate language to provide a basis for comparison with sub-transitive CFA. We have some

Bmrk	**NVars**	**Type Tree**	Type-Driven			Demand-Driven			Our Algo		
			$Node_{tot}$	$Time_{tot}$	Ratio	$Node_{tot}$	$Time_{tot}$	Ratio	$Node_{tot}$	$Time_{tot}$	Ratio
CM	14456	1397834	∞	∞	∞	∞	∞	∞	41742	$\star$	$\star$
CML	10289	16279	62116	14.47	23.42%	∞	∞	∞	24776	6.75	12.8%
BHut	2723	11641	29859	6.9	47.9%	6155	1.15	13.2%	6071	1.31	14.8%
Boyer	4961	1062	7227	3.49	27.6%	6468	2.83	23.4%	6466	2.80	23.1%
Life	1272	736	2843	0.44	21.15%	2461	0.34	17%	2415	0.38	18.5%
K-Bend	1855	689	4019	0.97	15.2%	∞	∞	∞	3378	0.52	9.1%
Lex	3205	1480	6280	1.10	10.4%	∞	∞	∞	4834	1.44	13.33%
Yacc	12314	33339	130878	$\star$	$\star$	∞	∞	∞	27120	9.09	14.2%
Vliw	8839	8162	21631	8.05	16.5 %	∞	∞	∞	19388	7.56	15.90%
Simple	4050	9123	21722	6.5	40%	14571	3.16	24.9%	14553	3.17	24.6%
Mbrot	621	69	1282	0.02	10%	909	0.02	10.5%	909	0.03	15%
Ray	1385	713	3563	0.54	16.4 %	2439	0.32	10.4%	2381	0.43	13.6%

Figure 10: Timing Results

tentative results, but no firm data to report here.

Our present system focuses on flow of functions and it provides only a trivial treatment of data structures (all data structure nodes are collapsed to one node). This has little impact on our analysis of functions (since functions are rarely put in data structures). However our desire is to provide a more generic analysis for reasoning about functions, data constructors and arithmetic. Our results so far indicate that a straightforward adaptation of the data structure rules from [9] is not feasible on large benchmarks. We are currently investigating alternatives.

Acknowledgement

We thank Prof. Zhong Shao for helping us with many of the FLINT queries. We also thank Chris League for providing some of the code we used for measuring type sizes.

References

[1] A. Aiken and E. Wimmers, “Type Inclusion and Type Inference”, *ACM Conference on Functional Programming and Computer Architecture*, 1993, pp 31–41.

[2] A. Bondorf and J. Jorgensen, *Journal of Functional Programming*, “Efficient analysis for realistic off-line partial evaluation”, Vol. 3, No. 3, July 1993.

[3] G. DeFouw, D. Grove, and C. Chambers, “Fast Interprocedural Class Analysis”, *Proc. 1998 ACM Symposium on the Principles of Programming Languages*, pp. 222–236, 1995.

[4] M. Fahndrich, J. Foster, Z. Su, and A. Aiken, “Partial Online Cycle Elimination in Inclusion Constraint Graphs”, *ACM Conference on Programming Language Design and Implementation*, 1998, pp 85–96.

[5] C. Flanagan and M. Felleisen, “Componential Set-Base Analysis”, *ACM Conference on Programming Language Design and Implementation*, 1997, pp 235–248.

[6] N. Heintze, “Set-Based Analysis of ML Programs”, *ACM Conference on Lisp and Functional Programming*, pp 306–317, 1994.

[7] N. Heintze, “Control-Flow Analysis and Type Systems”, *Static Analysis Symposium*, 1995, pp 189–206.

[8] N. Heintze and D. McAllester, “On the Cubic-Bottleneck of Subtyping and Flow Analysis” *IEEE Symposium on Logic in Computer Science*, 1997.

[9] N. Heintze and D. McAllester, “Linear-Time Subtransitive Control Flow Analysis” *ACM Conference on Programming Language Design and Implementation*, 1997, pp 261–272.

[10] S. Jagannathan and S. Weeks, “A Unified Treatment of Flow Analysis in Higher-Order Languages”, *Proc. 1995 ACM Symposium on the Principles of Programming Languages*, 1995.

[11] N. Jones, “Flow Analysis of Lambda Expressions”, *Symp. on Functional Languages and Computer Architecture*, pp. 66-74, 1981.

[12] N. Jones, “Flow Analysis of Lazy Higher-Order Functional Programs”, in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (Eds.), Ellis Horwood, 1987.

[13] R. Milner, M. Tofte and R. Harper, “The Definition of Standard ML”, MIT Press, 1990.

[14] Z. Shao, C. League, and S. Monnier, “Implementing Typed Intermediate Languages”, *ACM Conference on Lisp and Functional Programming*, pp. 313–323, 1998.

[15] O. Shivers, “Control Flow Analysis in Scheme”, *Proc. 1988 ACM Conf. on Programming Language Design and Implementation*, pp. 164–174, June 1988.

[16] B. Steensgaard, “Points-to Analysis in Almost Linear Time”, *Proc. 1996 ACM Symposium on the Principles of Programming Languages*, pp. 32–41, 1996.

[17] J. Y. Girard, ”Interpretation Fonctionnelle et Elimination des Coupures dans l’Arithmetique d’Ordre Superieur”, 1972.

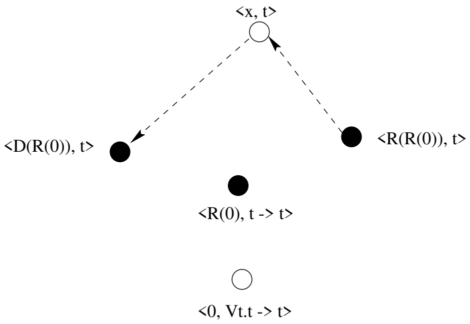
[18] John C. Reynolds, ”Towards a theory of type structure”, *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pp. 408–425, 1974.

A An example graph

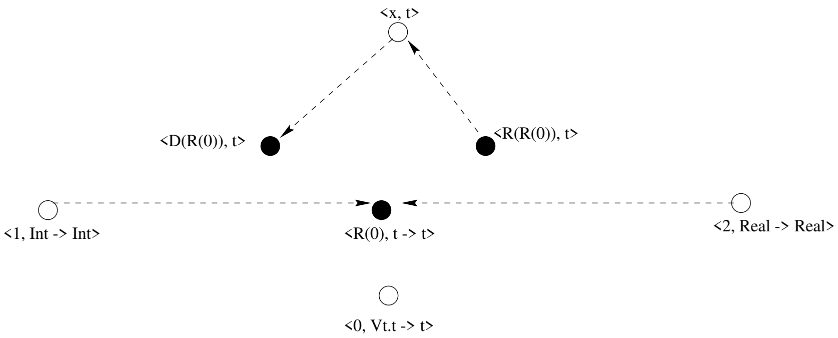
We construct the flow-graph for the following code fragment.

$$
\begin{aligned}
v_0 &= \Lambda t.\lambda x : t.x \\
v_1 &= v_0[Int] \\
v_2 &= v_0[Real] \\
v_3 &= @v_1 1 \\
v_4 &= @v_2 3.4 \\
v_5 &= (v_3, v_4) \\
v_6 &= \pi_1 v_5
\end{aligned}
$$

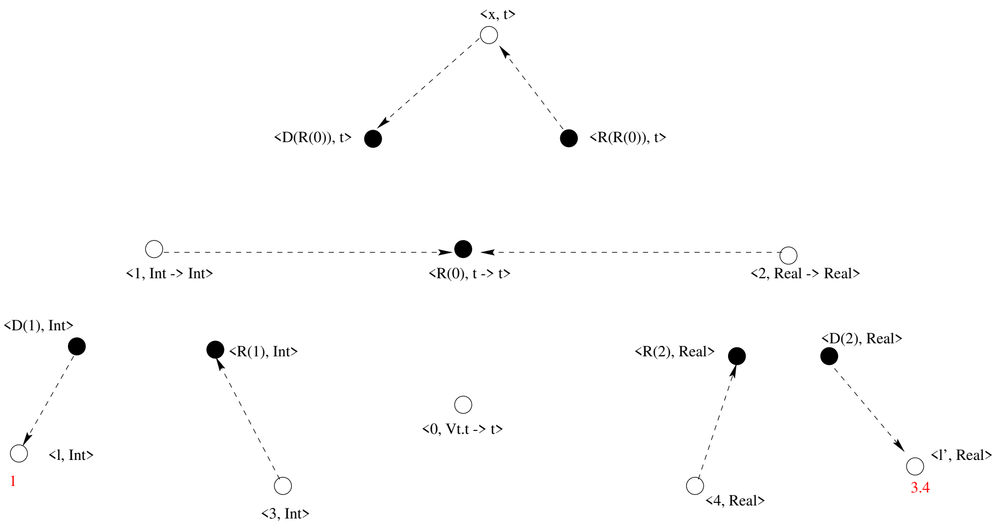
The variable bound to each expression is considered as the label for that expression. In the figures, the variable v_i is written as i. We use D, R, P_i to denote the domain, range and projection nodes respectively. Moreover, the administrative nodes are colored black. The dashed arrows are the arrows added during the first phase, the solid arrows are added during the closure phase. The arrows labelled *path* trace the path to the final result. Each node is represented as a tuple with the second field giving the type.



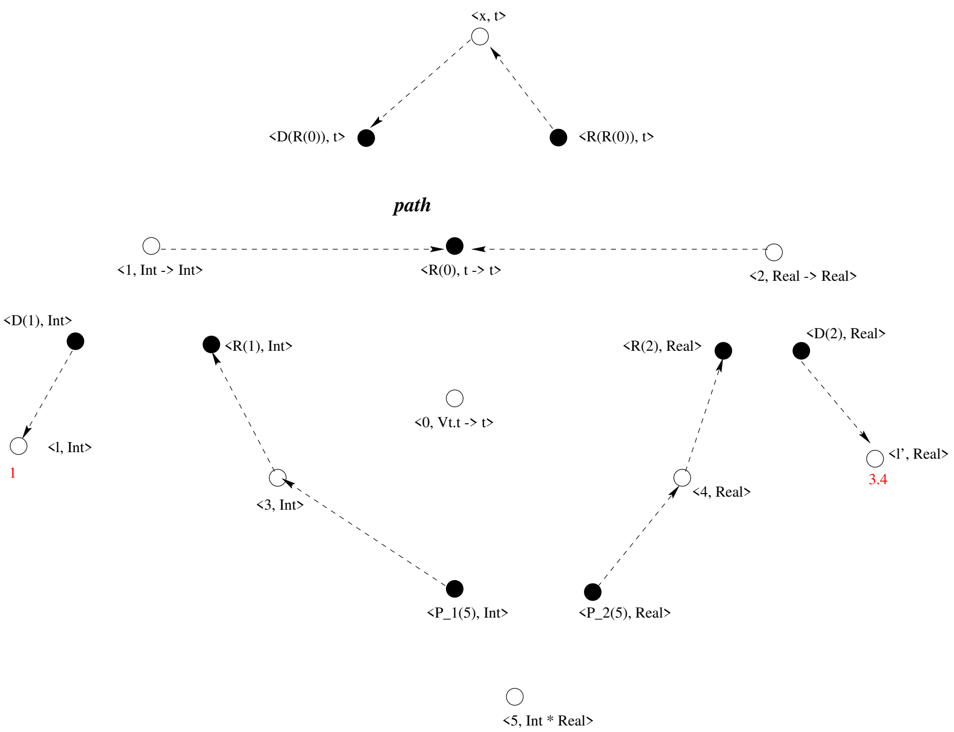
The Graph after v_0



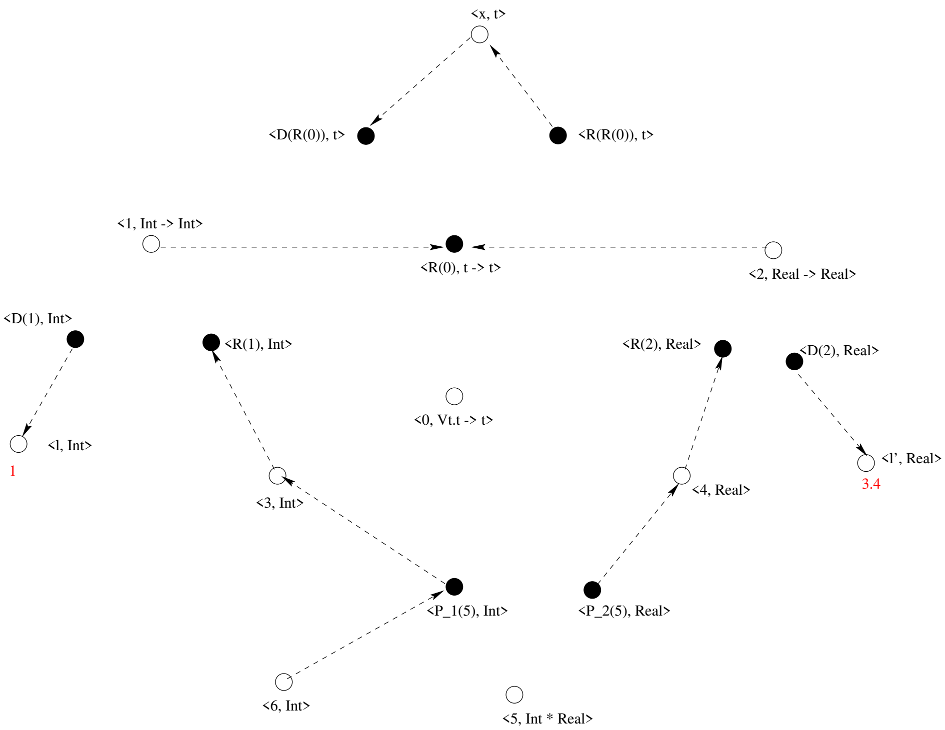
The Graph After The Type Applications



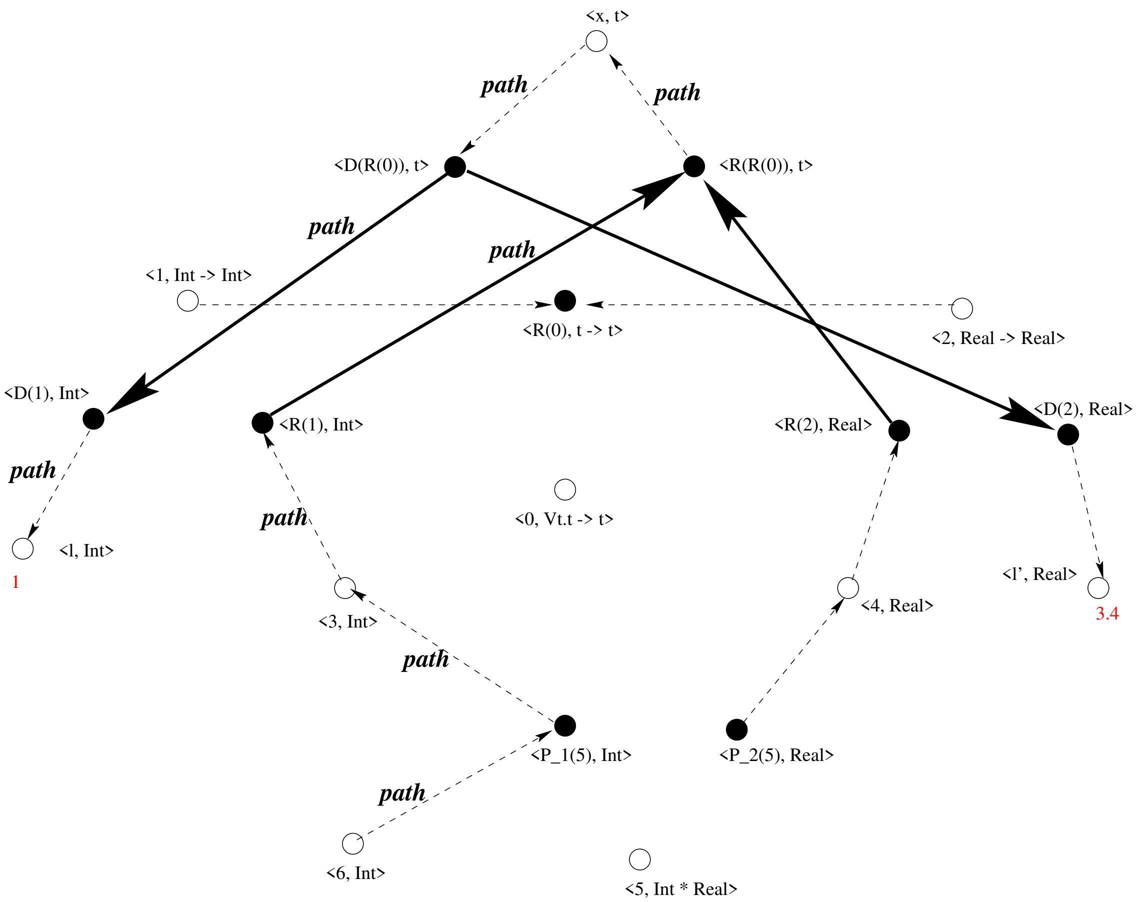
The Graph After The Applications



The Graph After Record Formation



The Graph After The First Phase



The Graph After Closure