18

digital
PARIS RESEARCH LABORATORY

An Abstract Machine to Implement Functions in LIFE

December 1992

Seth Copen Goldstein

PRL TECHNICAL NOTE

18

An Abstract Machine to Implement Functions in LIFE

Seth Copen Goldstein

December 1992

Publication Notes

This work was done during the author's three-month internship at Digital Equipment Corporation's Paris Research Laboratory.

Abstract

This note outlines the Life Abstract Machine (LAM), an abstract machine used as an intermediate target for the efficient compilation of LIFE. LAM focuses primarily on the efficient implementation of matching, residuation, and currying of functions. Although the topic is not discussed in this note, LAM also implements lazy unification.

LAM should be viewed as an intermediate target for compiling LIFE to a native instruction set for a general purpose processor. Thus, this note presents LAM as an abstract machine along with its instructions. However, we also discuss how LAM would be realized in terms of data structures and basic routines. This should facilitate the implementation of both a LIFE-to-LAM compiler and a LAM-to-native-code compiler.

Keywords

Logic programming, inheritance, functional programming, constraint logic programming, object-oriented programming, abstract machine, execution model, constraint, implementation, feature structures, LIFE.

Acknowledgements

Working at PRL has been a great experience and I want to thank everyone at the lab for a summer full of learning and fun. In particular I want to thank everyone in the LIFE group for keeping me on my toes, teaching me new things, and tolerating my bad French.

I have special thanks for Peter Van Roy, my advisor at PRL. Besides introducing me to LIFE and stimulating me with more ideas than I could digest, he kept me on course. Without his support and careful (and continual) reviews, this note and the work it contains would not have been completed.

In addition to working with Peter, it was a true pleasure to work with Hassan Aït-Kaci and Andreas Podelski. Between the two of them I learned a great deal about the theoretical nature of LIFE. Working with them on order-sorted feature theory unification was a truly enlivening process.

Finally, I want to thank Deborah for, well, everything.

Contents

1 Introduction

1.1 Overview

This note outlines the Life Abstract Machine (LAM), an abstract machine used as an intermediate target for the efficient compilation of LIFE. LAM focuses primarily on the efficient implementation of matching, residuation, and currying of functions. Although the topic is not discussed in this note, LAM also implements lazy unification.

LAM should be viewed as an intermediate target for compiling LIFE to a native instruction set for a general purpose processor. Thus, this note presents LAM as an abstract machine along with its instructions. However, we also discuss how LAM would be realized in terms of data structures and basic routines. This should facilitate the implementation of both a LIFE-to-LAM compiler and a LAM-to-native-code compiler.

While this document is intended to be self-contained, it is assumed that the reader is familiar with at least the informal parts of [2]. After presenting the basic properties of ψ-terms we give a brief of description of matching, residuation, and currying in Section 2. Section 3 explains how function definitions are decomposed into constraint trees. In Section 4 we present an overview of the data structures used for describing the algorithms that implement the functionality of LAM. The mechanics of currying and residuation, in particular for the equality constraint, are described in Section 5. In Section 6 the actual register set and instructions of LAM are described in detail. This is followed with some example LAM code in Section 7. Finally, the nitty gritty details of the implementation are described in Section 10.

1.2 ψ-terms

A ψ-term is a generalization of record-like data structures in traditional programming languages. It is an extension of first-order terms to include sorts and features. For a coherent and complete discussion of ψ-terms the reader can see [2]. Here I will briefly outline the notation.

A *ψ-term* (or *OSF*-term in normal form) is of the form $\psi = X : s(\ell_1 \Rightarrow \psi_1, \ldots, \ell_n \Rightarrow \psi_n)$ where

- there is at most one occurrence of a variable Y in ψ such that Y is the root variable of a non-trivial *OSF*-term (*i.e.*, different than $Y : \top$);
- s is a non-bottom sort in $\mathcal{S}$;
- $\ell_1, \ldots, \ell_n$ are pairwise distinct features in $\mathcal{F}$, $n \geq 0$;
- $\psi_1, \ldots, \psi_n$ are *normal OSF*-terms.

The sorts of a ψ-term live in a lattice. The least sort is bottom ($\perp$), the highest (or most general) is top ($\top$). If a sort, a, is below another sort, b, then we say that a implies b, a entails b, a is more specific than b, or equally, that a is subsumed by b[1]. Thus, all sorts are subsumed by $\top$ and $\perp$ implies every sort. Intersection of sorts is carried out by the greatest lower bound

[1]The details of sort implication and in particular how they relate to function invocation in LIFE can be found in [1]

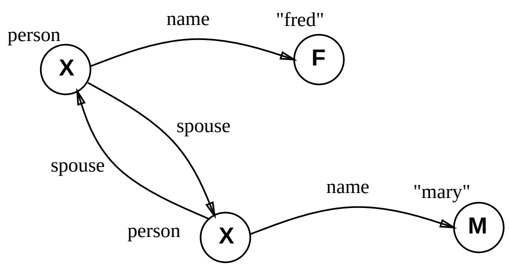


Figure 1: *Graphical representation of a ψ-term. The nodes represent sorts, and the arcs features. The capital letters in the nodes correspond to the variables used in the textual representation of the ψ-term.*

operator, written, $\wedge$.

ψ-terms can be represented in many ways. The first two considered here are the textual and graphical representations. For example, the ψ-term

$$\begin{aligned} &X\!:\!person(name \Rightarrow F\!:\!\text{“fred”},\\ &\qquad spouse \Rightarrow S\!:\!person(name \Rightarrow M\!:\!\text{“mary”},\\ &\qquad\qquad\qquad spouse \Rightarrow X))\end{aligned}$$

Can also be represented by the directed graph in Figure 1. The graphical representation does not actually need the variable names, used in the textual representation to capture equality constraints. However, to aid in referencing the nodes and arcs we will keep them. An alternative representation of ψ-terms is the *OSF*-clause. An *OSF*-clause is a conjunction of a *OSF*-constraints. An *OSF-constraint* is one of (1) $X : s$, (2) $X \doteq X'$, or (3) $X.\ell \doteq X'$, where X and X' are variables in $\mathcal{V}$, s is a sort in $\mathcal{S}$, and ℓ is a feature in $\mathcal{F}$. An *OSF-clause* is either an *OSF*-constraint or of the form ϕ & ϕ' where ϕ and ϕ' are *OSF*-clauses. We can read $X : s$ as “X lies in sort s”, $X \doteq X'$ as “X is equal to X'”, and $X.\ell \doteq X'$ as “X' is the feature ℓ of X.”

We can always associate with an *OSF*- term $\psi = X : s(\ell_1 \Rightarrow \psi_1, \ldots, \ell_n \Rightarrow \psi_n)$ a corresponding *OSF*-clause $\phi(\psi)$ as follows:

$$\begin{aligned}\phi(\psi) = \quad & X : s \;\&\; X.\ell_1 \doteq X'_1 \;\&\; \ldots \;\&\; X.\ell_n \doteq X'_n\\ & \&\; \phi(\psi_1) \qquad \&\; \ldots \;\&\; \phi(\psi_n)\end{aligned}$$

where $X'_1, \ldots, X'_n$ are the roots of $\psi_1, \ldots, \psi_n$, respectively. We say that $\phi(\psi)$ is obtained from *dissolving* the *OSF*-term ψ. For example, the ψ-term in Figure 1 could be represented by four sort constraints, $X\!:\!person$, $F\!:$“fred”, $S\!:\!person$, and $M\!:$“mary”, and three feature constraints, $X.name \doteq F$, $X.spouse \doteq S$, $S.name \doteq M$, and $S.spouse \doteq X$. While an *OSF*-clause is just a conjunction of the primitive constraints, it can also be represented by a tree which shows how the individual constraints are related to each other. Figure 2 shows the above ψ-term's constraint tree. In LIFE the function call is represented by a ψ-term, where the sort of the ψ-term is the name of the function being invoked, and the arguments to the function are ψ-terms connected to the root by features that label them as arguments. For instance, the function call `append([1,2,3], [4,5])` is really shorthand for `append(1` $\Rightarrow$ `[1,2,3], 2`

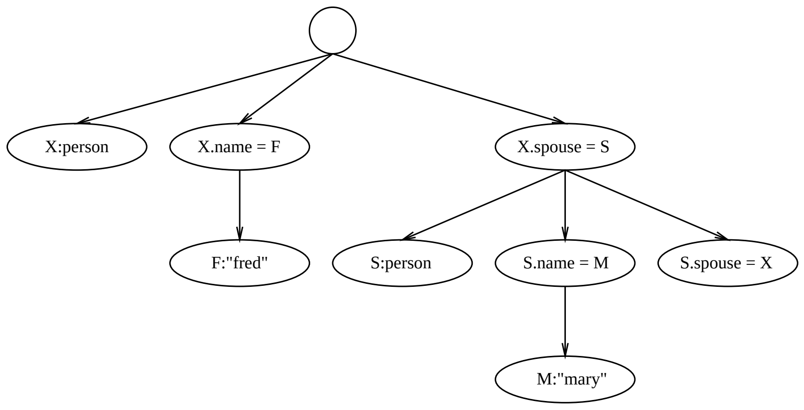


Figure 2: The Constraint Tree representation of the dissolved ψ-term.

$\Rightarrow$ `[4,5])`. As will be seen later, representing the function calls and function definitions as constraint trees leads to a natural and efficient compilation strategy.

2 Matching

2.1 Definition

Function invocation in LIFE is accomplished with matching. As a result the rules for invoking functions in LIFE follow the rules of implication, which means there are three cases that need to be considered when implementing function invocation: (1) when the actual arguments imply (*i.e.*, entail) the formal arguments, (2) when the actuals imply the negation of the formals (*i.e.*, they *disentail* the formals), and (3) when the actuals neither entail nor disentail the formals. In the last case we say that the function invocation has *residuated* on its arguments. The requirements for an implementation of function invocation in LIFE are that a function invocation either fire or fail as soon as the actuals either entail or disentail the formals and that no changes be made to the arguments unless and until the function fires[2]. Finally, the implementation must maintain the church-Rosser property of functions. The main challenge to the implementor is to perform the smallest possible number of checks. This section describes an efficient implementation which will perform each check only once regardless of the number of times a function residuates or its arguments are lowered. There are two components to this implementation: an abstract machine and a compilation strategy. Most important, it will also handle the most common cases, that of direct entailment or disentailment, with surprising efficiency.

2.2 Examples

In order to point out some of the difficulties in implementing functions, let's assume the definitions below and the sort hierarchy in Figure 3.

[2]A formal treatment of functions in LIFE can be found in [1].

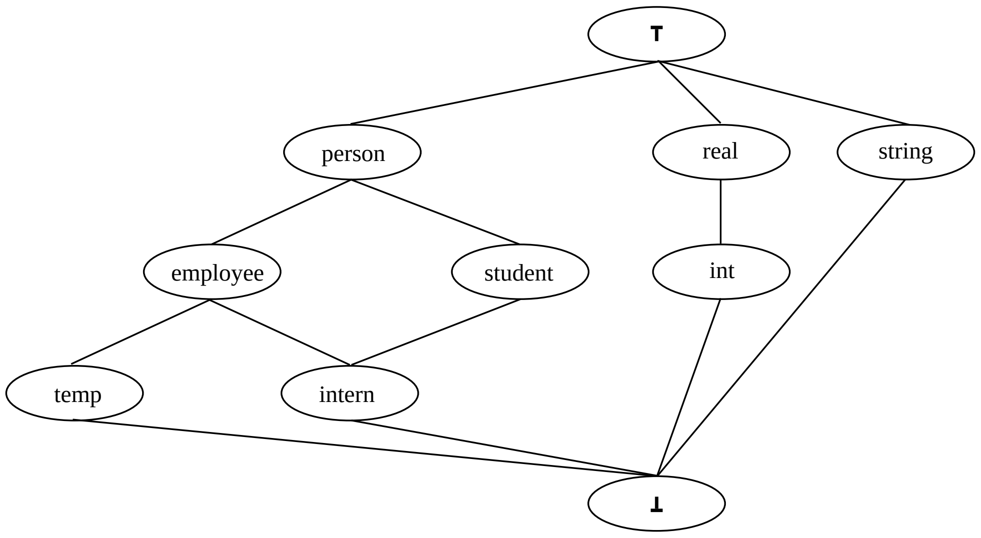


Figure 3: *A sample sort hierarchy.*

```
incr(X:int) → X+1.
incr(X:string) → "increment".
insurance(P:person(spouse ⇒ S:temp)) → 1.
insurance(P:person(spouse ⇒ S:person)) → 2.
isspouse(P:person(spouse ⇒ S), S:person(spouse ⇒ P)) → true.
```

If `incr` is called with an integer or string argument we get a result. For example (the answer produced follows the ⤳):

```
A=incr(5)?                    ⤳  A=6
A=incr("hello")?              ⤳  A="increment"
```

In addition to these obvious results, if an argument to `incr` is incompatible with the formal definitions, then it will result in failure.

```
A=incr(person)?               ⤳  failure
```

The interesting case is where the argument in the function call is under-specified, *i.e.*, its sort is neither incompatible nor a subsort of the formal argument. For example,

```
A=incr(X:@)?                  ⤳  A=@, X=@
X=real?                       ⤳  A=@, X=real
X=7?                          ⤳  A=8, X=7
```

In this example, X starts off being the top sort (represented by the character '@'), so the function `incr` residuates on its argument. When X is lowered to the sort real, the function is awakened and (because the sort of X is still to general) again residuated. When X is finally lowered to the integer 7, the function is reactivated and fires.

The function `insurance`, like `incr`, has two clauses in its definition. But it differs from `incr` in that the formal argument in the second clause is not incomparable with that in the first clause. Instead, it is more general than the one in the first clause.

```
A=insurance(X:person)?              ⤳  A=@, X=person
X=person(boss ⇒ "joe")?            ⤳  A=@,
                                        X=person(boss ⇒ "joe")
X=person(spouse ⇒ Y:person)?       ⤳  A=@,
                                        X=person(boss ⇒ "joe",
                                                 spouse ⇒ Y),
                                        Y=person
```

In this example, the function residuates not because the root sort of the argument is under-specified, but rather because the argument is missing a feature term. Notice that when the argument X is lowered by adding the feature `boss` it does not affect the residuation. When it is lowered again, by adding the feature `spouse`, the function remains residuated even though the second clause is satisfied. This brings out the point that before the next clause is tried the current clause must be completely disentailed.

To continue this example,

```
Y=intern?                           ⤳  A=2,
                                        X=person(boss ⇒ "joe",
                                                 spouse ⇒ Y)
                                        Y=intern
```

We see that once the first clause is disentailed, because $intern \wedge temp \doteq \perp$, the second clause is checked and in this case fires. This example also shows how equality constraints must be considered in disentailment.

```
A=isspouse(X:person, Y:person)?     ⤳  A=@,
                                        X=person,
                                        Y=person
Z=person(boss ⇒ "joe"), X=person(spouse ⇒ Z)?
                                    ⤳  A=@,
                                        X=person(spouse ⇒ Z),
                                        Y=person,
                                        Z=person(boss ⇒ "joe")
Y=person(boss ⇒ "fred")?           ⤳  failure
```

Up until the last query Z, the spouse of X, was unifiable with Y. However, in the last step Z and Y became incompatible and thus the original query had to result in failure.

Although the arguments to a function call cannot be modified in the matching process, information needs to be propagated in the ψ-term between constraints. For example, assume the sort hierarchy in Figure 4 and the following function definition: `theSame(X, X, X)` $\rightarrow$ `1.` The call `theSame(A:a, B:b, C:c)?` must fail immediately. This is because the ψ-terms A, B, and C can never unify.

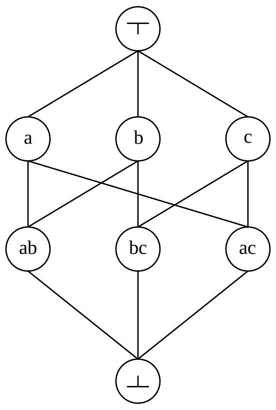


Figure 4: *A sample sort hierarchy.*

2.3 Introduction to Currying

The final difficulty in implementing functions in LIFE is that function calls can curry. If a function call does not have all its arguments specified in the function definition, then executing the function returns a ψ-term that is a curried function call. It does not return a ψ-term that stands for the result of the function call. This curried function call is a first-class object. If the rest of the arguments are added to the curried function call, then the function will execute. Thus, any unification performed on the ψ-term representing the curried function will not affect the result of the function call, but rather will add arguments to the call. For example,

```
A=incr?                      ⇝  A=incr
A=@(1 ⇒ X:@)?                ⇝  A=@, X=@
A=@(extra ⇒ 2)?              ⇝  A=@(extra ⇒ 2), X=@
X=5?                         ⇝  A=6(extra ⇒ 2), X=5
```

In this series of queries, `A` is first set equal to the curried function `incr`, then the first and only argument is unified with `A` and then `A` becomes the result of the function call, which has residuated on `X`. An extra feature is then added to the *result* of the function call. Finally, `X` is lowered and the function completes. Notice that the order of the second and third line can*not* be changed, because in the second query `A` represents the call and in the third it represents the result!

3 Representing Function Definitions as Constraint Trees

In order to obtain the behavior described above, this note proposes a compilation scheme based on decomposing function definitions into function trees, where each clause of the function is represented by a constraint tree[3].

[3]The idea of using executable constraints was found in [4]

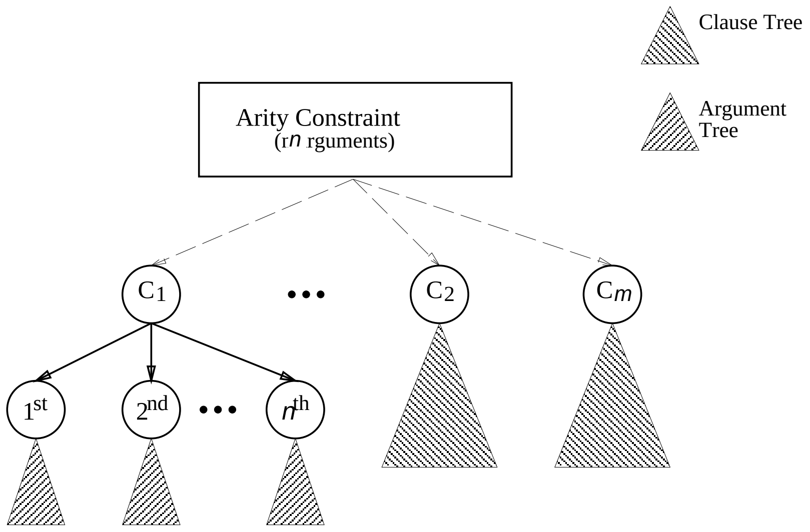


Figure 5: *Tree representing an* m-*clause* n-*ary function definition.*

Each function is defined by a (possibly unit) series of clauses. Each clause is tried in turn until either all are disentailed or one is entailed. If a particular clause residuates, then it is only when it is disentailed that the remaining clauses are tested.

Each function head, or clause, is compiled into a series of constraints. In this note, the individual clauses are joined together into a complete function definition in the most basic way—serially—through the use of the failure mechanism. It is expected that optimization techniques can be performed on the individual clauses to create a more efficient conglomeration. One way to view a function definition is as a tree. The root of the tree will hold an arity constraint; which succeeds iff the actual function call has the same number of arguments as in the definition. The function represented in the tree in Figure 5 is an n-ary function. The children of the arity constraint node represent each clause in the function definition. The clauses are ordered from left to right. Each clause then has *n* children, representing the constraint tree needed to do the matching for each of the n arguments of the function. These constraint trees are created by dissolving the ψ-terms that represent each argument in the clause. It is similar to the constraint tree in Figure 2. When a failure is detected, then control is transferred to the next clause node in the tree. The advantages in viewing a function definition this way are many. First, the wavefront algorithm [4, 6, 5] as implemented in this note treats the constraints on the arguments as if they were arranged in a tree. Second, one can classify the arcs and nodes in such a way as to guide optimizations. For instance, in Figure 5, the solid arcs can be rearranged in any order, while the dashed arcs are fixed. It is also worth noticing that since the arity constraint is the same for all the clauses of a definition, it might be inlined into the calling code, so that ψ-terms don't have to be built for function calls.

4 Data Structures

This section gives an overview of the data structures used in describing the LAM instruction set. The basis of the machine is the representation of ψ-terms. The class `Psiterm` defined here is used to express the parts of a ψ-term that concern matching.

```
class Psiterm
{
    Sort            sort;           // the sort for this Psiterm
    Features        features;       // the features
    Residuation     rlist;          // residuations for this Psiterm
    Psiterm*        ref;            // the dereference chain link
};
```

`sort` is the sort that the ψ-term currently has. The field `features` points to all the features in the ψ-term. Its internal representation does not concern us here. Any collection that supports fetching, adding, and testing the existence of a feature will suffice. The most interesting field in terms of matching is the `rlist` field; it points to a `Residuation` which is described below. The `ref` field is used to implement dereference chains. If it is not NULL, then the ψ-term has been unified to the ψ-term pointed to by `ref`.

Every function invocation is associated with a `Frame`. The relevant parts are

```
class Frame
{
  int      residCounter;  // counts number of residuated variables
  Code     body;          // address of body of function
  Code     fail;          // the address to goto if a constraint fails
  Psiterm* result;        // result of the function gets put here
};
```

Missing from the above definition is any information about local variables and so forth that every function frame will hold. This definition is just sufficient for describing the matching process. The `residCounter` is initialized on entry to zero and every residuated goal increments this counter. The instruction **resid?** checks this field. If it is zero, then the function body, pointed to by the field `body`, is executed. The `fail` is a pointer to the next clause in the function definition to be executed if any of the constraints in the current clause tree fail. If the current clause tree is the last, then this will point to code that will invoke the general backtracking routine. The ψ-term that is returned by the function is pointed to by the `result` field. When a constraint residuates it creates (if necessary) a `Residuation` which is attached to the ψ-term that caused the residuation.

```
class Residuation
{
  Frame*       parent;           // frame of function to be activated
  Sort         sort;             // the best known sort for this term
                                 // (i.e. the glb of the formal and the
                                 // actual)
  Residuation* next;             // next residuation for this var
  ResidInfo*   info;             // info about each resid for this parent
};
```

Each ψ-term can have only one `Residuation` per frame in which it has residuated. Each constraint that it residuates on is pointed to by the `info` field of the `Residuation`. Sort constraints, feature constraints, and initialized constraints all create `ResidInfo` instances. Equality constraints create an instance of `EqResidInfo`. In addition to its use for residuation, `ResidInfo` is also used when functions curry.

```
class ResidInfo
{
```

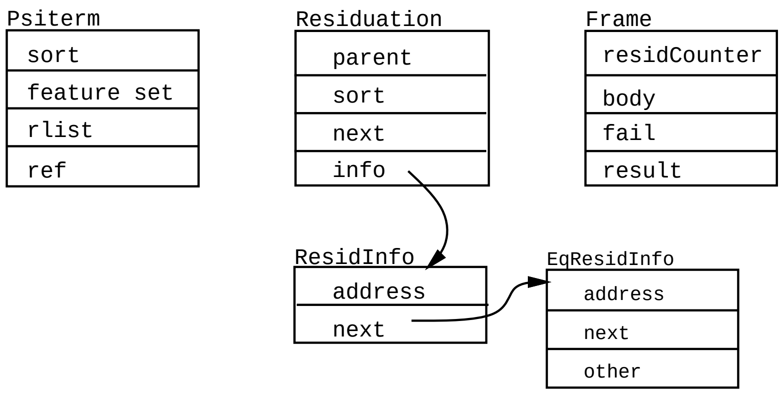


Figure 6: *Representation of the basic data structures.*

```
      Code        address;        // address of constraint to re-execute
      ResidInfo   next;           // next ResidInfo for this frame if it
                                  // exists, else NULL
    };

    class EqResidInfo: public ResidInfo
    {
      Residuation other;          // Residuation for the other psiterm
                                  // used in = constraint
    };
```

In what follows we will represent the data structures by schematic diagrams. The diagrams will not name the fields of the data structures, but will just stack the fields upon each other as in Figure 6.

5 The Mechanism Behind the Machine

5.1 The Phases of a Function Call

The execution of a function has several phases. The first is upon initial entry to the function. In this phase the arity constraint (see Section 6.3) is executed. If the arity constraint succeeds, the ψ-term representing the function call is deconstructed and the execution of the function passes into the matching phase; otherwise the function call is curried and immediately returns. The deconstruction of a ψ-term is just placing the arguments into registers by tracing the features from the original function ψ-term.

When the function has passed into the matching phase a frame will be allocated to the function. Every function will have frames with at least the structure described above for class `Frame`. In addition the frames, will have fields for any variables that are manipulated in the function.

In the matching phase, the constraints that result from the function definition are executed in order to find the clause of the function definition. If any constraint fails, then the next clause is tried, until no clauses remain, at which point the function invocation fails. If any

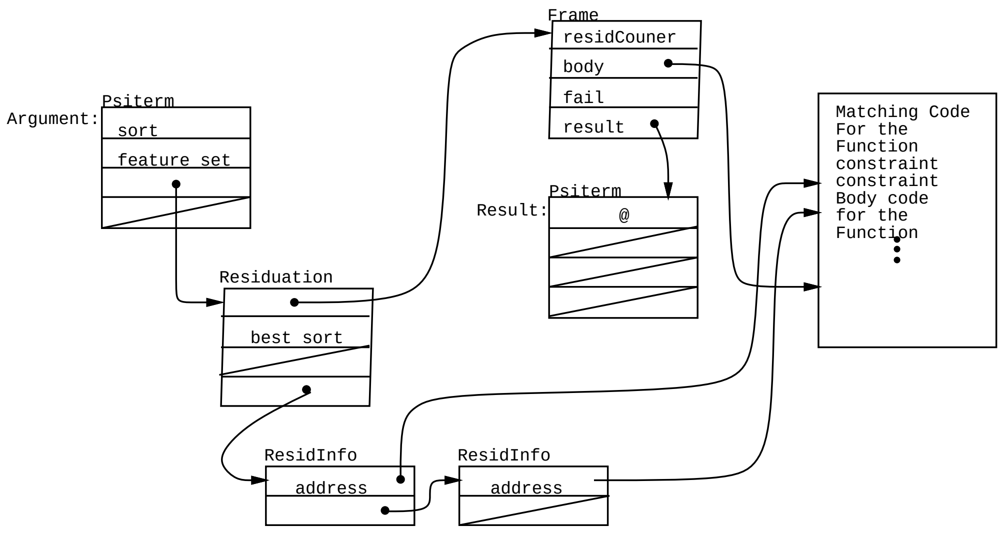


Figure 7: Schematic representation of a ψ-term with two residuations attached for the same function invocation.

clause completes with success and none of the constraints have residuated, then the clause that matched enters the execution phase and executes the body of the function.

If a constraint residuates during the matching phase, a `Residuation` is created which is attached to the ψ-term involved. The remaining constraints—the ones below the residuating constraint in the tree—will be skipped. If all of the executed constraints either succeed or residuate the function becomes quiescent and returns to the caller.

If any of the ψ-terms that have residuations are later modified, then the residuations will resume execution to recheck the constraint that caused the residuation during the matching phase. This is called the resumption phase.

5.2 Residuation

When a constraint in the head of a function definition residuates, a series of objects is created and attached to the ψ-term that caused the constraint to residuate. The objects created have two primary functions: first, to force disentailment if necessary, and second, to allow the constraint to be resumed if the ψ-term is modified.

In order to keep the amount of information stored in each residuation to a minimum, the residuation structure is broken down into two objects: a `Residuation` object, which records the information about the frame on which the ψ-term is residuated, and a `ResidInfo` object, which records the particular constraint that residuated. In this way, if a single ψ-term is involved in multiple residuated constrains for a particular frame, only multiple ResidInfo objects need to be created. See Figure 7 for an example of a frame with a ψ-term that has residuated twice.

Before the residuation is created, the sort of the ψ-term is also compared to the sort field in the Residuation structure. If they are incompatible then the function definition is disentailed.

In this manner, information about equality between the actual arguments is maintained even though the arguments themselves cannot be modified during the matching phase[4].

The three simple constraints (subsort, feature existence, initialized) each create a `ResidInfo` which is attached to the corresponding Residuation object for the ψ-term and frame involved. The equality constraint, on the other hand, must create a structure so that if *either* object in the constraint is modified, then the constraint will be re-executed. Further, if either of the ψ-terms involved in the equality constraint contains subparts that cannot be unified, then the constraint must generate a failure.

This added complexity is handled by having the constraint traverse both ψ-terms, adding `EqResidInfo` objects to all common children. For example, in Figure 8 the ψ-terms attached to the features in common to A and V must each be checked to see if possible conflict (yielding failure) or possible unification (yielding more EqResidInfo structures) results (also see the example on Page 47). Notice how each pair of corresponding ψ-terms involved records the sort of their intersection. Further, if a new feature is added to a ψ-term involved in an equality residuation, additional work will occur only if there is a corresponding ψ-term involved (*i.e.*, it already has an EqResidInfo attached).

Thus, for each frame in which a ψ-term has any residuations, there will be one `Residuation` structure attached to the ψ-term. For each constraint (in the same frame) on which the ψ-term residuated there will be either a `ResidInfo` or a `EqResidInfo` structure linked to the `Residuation` structure. For example, if n ψ-terms were involved in $n-1$ equality constraints, all of which residuated, there would be n `Residuation` structures–each attached to one of the n ψ-terms. In addition, there would be $(n-1)$ pairs of `EqResidInfo` structures–one for each constraint–linking up the ψ-terms that were involved in the $n-1$ equality constraints.

5.3 Resumption

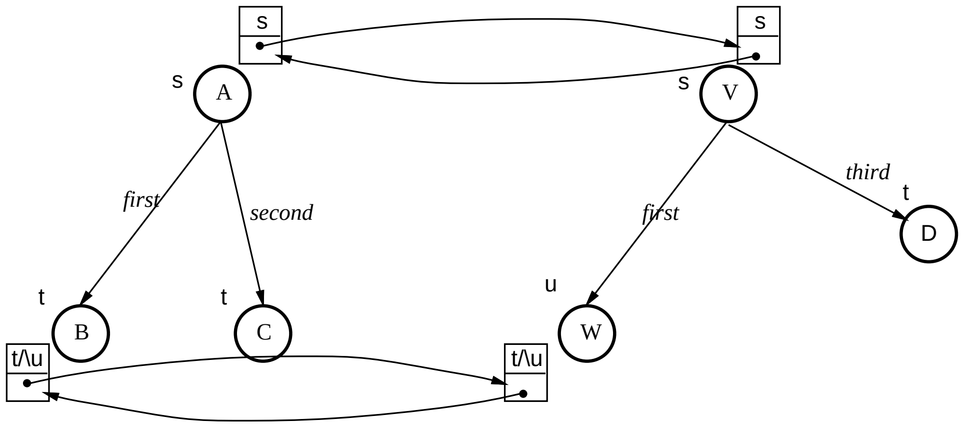
When a ψ-term is modified the unification routine will check to see if there are any residuations depending on the modified ψ-term. If there are, then each residuation is executed in turn. If any of the functions becomes satisfied, then it in turn continues to execute.

In other words, if during unification of a ψ-term, a function that was residuated on the ψ-term is enabled, it will be executed immediately. Thus, the enabled function runs (and either fails or completes) before continuing with the code that caused the unification to happen in the first place.

5.4 Currying

If an attempt is made to execute a function without all of its arguments, then the function will curry and return. At a later time the extra arguments may be unified with the function call and then the function will execute. As currying is not a common occurrence, we strove to

[4]It was recently noted that storing the sort, in and of itself, is not sufficient to provide complete disentailment. For instance, assume the function definition: `f(s(a), s(b)) -> 1.`. If $a \wedge b$ is $\perp$, then the call `f(X, X)` should fail. However, since our model only compares the sorts of residuated ψ-terms, it will not cause failure. This can be fixed by following the model of the equality constraint.



The EqResidInfo structures that would be created by executing the equality constraint on ψ-terms *A* and *V*.

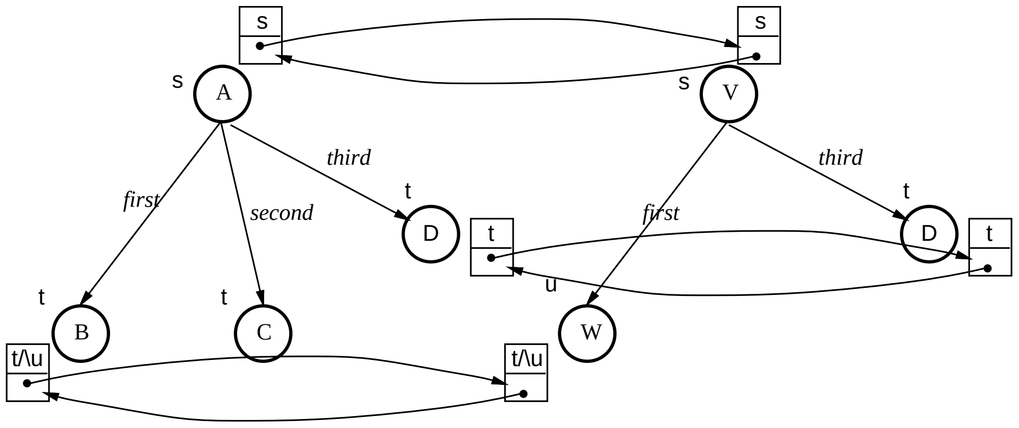


Figure 8: *The additional structures that would be added if the* third *feature were added to ψ-term* A.

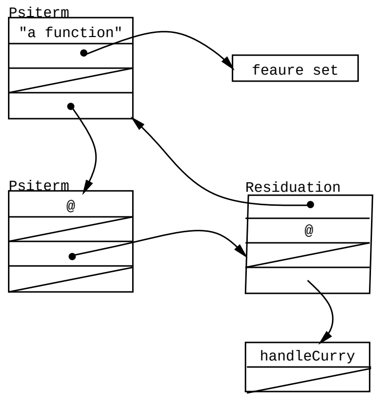


Figure 9: *This figure shows the result of executing the arity constraint when the function needs to be curried.*

allow currying without increasing the cost of doing general operations like unification. This is achieved by treating currying like residuation.

If a function curries, a new ψ-term, called a *curry term*, will be created with sort top and no features. The curry term will have an attached ResidInfo which has a pointer to the `handleCurry` routine (see Figure 9). Further, the original ψ-term will be "unified" with the newly created curry term in the sense that dereferencing the original ψ-term will return the new curry term. Furthermore, when the newly created ψ-term is unified with the special curry term, the old ψ-term will will have its arguments copied into the new term. This allows the curry term to represent a closure and be used as many times as the curried function is applied. Thus, if extra arguments are added to the original function call the unification procedure will unify the extra arguments with the curry term. Since the curry term has no features and is of sort top the unification will always succeed, invoking the residuation attached to the curry term in the process. The residuation function invoked will always be the `handleCurry` routine.

The `handleCurry` routine will first break the dereference link between the original function and its curry term. It will then copy the original term to the curry term and then it will then try to unify the curry term with the ψ-term representing the original function call. If the unification succeeds, it will retry the function. Otherwise the unification will fail in the normal way. Thus, the mechanism for residuation handles currying without any extra overhead.

6 LAM: The Life Abstract Machine

The LAM is an abstract machine used as the intermediate target for the compilation of LIFE programs. It includes mechanisms that directly facilitate the compilation of LIFE features. This section describes the registers and instruction set of LAM. While a complete abstract machine would have to implement all LIFE functionality (*i.e.*, residuation, unification, choice-point handline, etc.), LAM–and this document–focus primarily on the mechanisms needed to handle function calls and residuation.

PC	Program Counter	
SP	Stack Pointer	
R*n*	General purpose register R*n*	There are an unlimited number of general purpose registers. They can each hold a pointer to a ψ-term, an integer, or any other basic data type. They are denoted as the letter R followed by a number, *e.g.* R8.
CFA	Current Fail Address	The CFA register holds the code address to jump to if a unification operation fails.
CF	Current Frame	The CF register contains a pointer to the frame of the function currently executing.
CPT	Current ψ-term	The CPT register by convention points to the root of the ψ-term being operated on.
RR	Result Register	The RR register points to the ψ-term that the last function returned. It is only used when arguments are passed in registers.

Table 1: *The LAM register set.*

6.1 The LAM Register Set

In addition to general-purpose registers, LAM includes registers used to control the process of unification and matching. All of these additional registers are used as pointers to data structures in memory. They are not special except that their use is predetermined, so that we can more easily describe the instruction set. By having them implicit in the instructions we are also able to pack more information into each instruction.

LAM makes no assumptions about the number of registers in the machine. It is assumed that the mapping from LAM to the machine language of a general purpose computer would map the LAM registers as efficiently as possible. LAM also assumes that no tags are available in memory or in the registers. Figure 1 lists all the registers in LAM important to this document.

6.2 The LAM Instruction Set

This section describes the instructions in the LAM instruction set. It focuses on the instructions needed to do matching. The instruction set is specified at a high level so as to allow latitude in the

LAM–to–native-instruction-set mapping. For instance, it does not specify how heap-allocated structures are managed. It does not take a position on the actual representation of integers in their boxed or unboxed form. Its primarily addresses the functionality of matching, residuation, failure, and unification. It ignores the handling of choice-points and the implementation of built-in operations.

In the instruction descriptions below operand names that begin with R denote any register. X is used to denote a memory location. Italicized names ending in *adr* refer to addresses in the code space; otherwise an italic name is a generic label. Feature names are represented by ℓ, and sort names are represented by s. Each instruction is followed by an informal description and pseudo-code that describes its function. The pseudo-code will often refer to data structures and routines defined in Sections 4 and 10.

Whenever an instruction operates on a ψ-term, it is assumed that the ψ-term has already been dereferenced at the time it is fetched. In the descriptions of the instructions the phrase "the ψ-term Y" (or, if the context is clear, just Y) is understood to mean "the ψ-term that is at the end of the reference chain from the ψ-term pointed to by Y."

As LAM instructions are already fairly high-level no real constraints are placed on the addressing modes that the operands may use, unless otherwise specified. The three basic modes used here and in the examples are register, register + offset, and memory. The register + offset mode is represented by writing R$\rightarrow$*offset*. Notice that this is distinct from R.ℓ, which retrieves the ψ-term reached by following the ℓ feature from the ψ-term pointed to by R.

The instructions are broken down into two categories: head instruction and body instruction. The head instructions are those instructions that can only appear in function heads. The distinction between head and body instructions are between those that modify ψ-terms and those that do not. Head instructions do not modify the value of ψ-terms, they only check there contents and sometime cause residuations to be associated with them. Body instruction, on the other hand, cause the ψ-terms they operate on to change value. The one exception to this is **deref** which can be used in either section. It is listed in the body instruction section.

We will use a four part format to describe all the instructions. First the name and syntax of the instruction will appear as follows.

$\boxed{\textit{instructionformat}}$ . . . . **instruction name**
After the instruction format will be a description of each operand, then a textual description of the function, and, finally, a pseudo-code description of the instruction's functionality.

6.3 Head Instructions

Each head instruction is a constraint that checks its operand, the actual argument to the function, against the function definition. The constraints are different from the body instructions in that they don't modify the operands and in that they can create residuations. Thus, while body constraint instructions will either succeed or fail, head constraints will either succeed, residuate, or fail.

All the head instructions have an implicit form, and some of them have an explicit form. The implicit form, to work correctly, requires the CF register and certain of the fields of the current frame to have been initialized for residuation and failure. On the other hand, the explicit version will transfer control to an explicitly named address if either residuation or failure occurs, so that nothing needs to have been preset in the current frame.

The implicit versions of the instructions will invoke the macro **headfail** if a failure occurs. This macro will check CF$\rightarrow$*fail*. If CF$\rightarrow$*fail* is nonzero, then control will be transferred to CF$\rightarrow$*fail*; otherwise control will be transferred to the address in CFA. This is the mechanism used to handle multiple rules in a function definition. The pseudo-code for the macro **headfail** is

```
headfail:
    if $\mathrm{CF}\rightarrow fail \neq 0$ then
        $\mathrm{PC} = \mathrm{CF}\rightarrow fail$
    else
        $\mathrm{PC} = \mathrm{CFA}$
    endif
```

$\boxed{\begin{array}{l}\mathrm{Rx}\subseteq s\\ \mathrm{Rx}\subseteq s\;\; recheckadr\\ \mathrm{Rx}\subseteq s\;\; failadr, residadr\end{array}}$ . . . . **Subsort Constraint**

Rx	Rx must points to a ψ-term.
s	*s* is a sort defined in the sort hierarchy.
recheckadr	*recheckadr* is the address of the re-evaluation routine.
failadr	*failadr* is the next address that will be executed if the constraint fails.
residadr	*residadr* is the next address that will be executed if the constraint residuates.

The *Subsort Constraint* instruction determines if the sort of Rx is a subsort of *s*. There are three possible outcomes for this instruction: success, failure, or residuation. If Rx$\rightarrow$*sort* is subsumed by *s*, then this constraint succeeds. If the Rx$\rightarrow$*sort* $\wedge$ *s* is $\perp$, then the instruction fails. If neither of these occur, or, in other words, Rx$\rightarrow$*sort* neither entails or disentails *s*, then the instruction residuates. See Figure 10 for the various possibilities.

The Subsort Constraint instruction has an implicit and explicit version. The implicit version is Rx$\subseteq$*s* *recheckadr*. If *recheckadr* is not specified, then it is set to the current PC. The explicit version is Rx$\subseteq$*s* *failadr*, *residadr*. Both the implicit and explicit versions check to see if the sort of Rx is compatible with *s* without altering Rx in any way. If it is compatible, then the next instruction executed is the subsequent one in the function definition.

The two versions differ in how they handle residuation and failure. If the instruction residuates, the implicit version will create a sort residuation which will be attached to Rx and will resume at *recheckadr*. The explicit version will transfer control to *residadr*. It will not create any residuation structures, nor will it increment the *residCount* in the current frame.

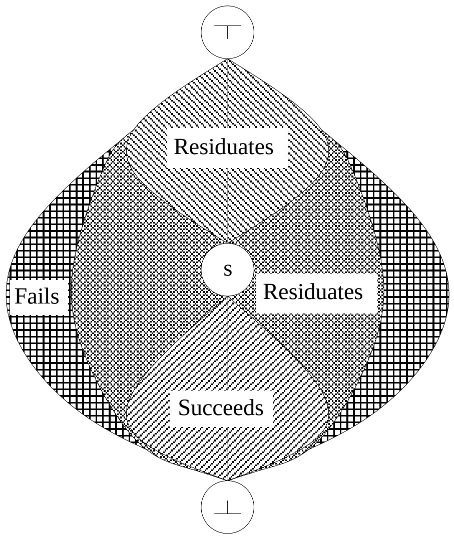


Figure 10: *Different regions of sort intersection.*

If the sort of Rx and *s* are incompatible, then the implicit version will execute the **headfail** macro. The explicit version will transfer control to *failadr*.

Description of Rx$\subseteq$*s* *recheckadr*:

```
s′= glb(Rx→sort, s)
switch (s′)
case ⊥ :
    headfail
otherwise :
    if (s′≤ s) then
        PC = PC + 1
    else
        Create a residuation and attach it to Rx
        Rx→ tryAddSimple(CF, s′, recheckadr)
    endif
endswitch
PC = PC + 1
```

Description of Rx$\subseteq$*s* *failadr*, *residadr*:

```
s′= glb(Rx→sort, s)
switch (s′)
case ⊥ :
    PC = failadr                                constraint fails
otherwise :
        if ($s' \leq s$) then
            PC = PC + 1                          constraint succeeds
        else
            PC = residadr                        constraint residuates
        endif
    endswitch
```

Rx.ℓ? Rx.ℓ? *residadr* Rx.ℓ? *residadr*, *recheckadr*	. . . . **Feature Existence**

Rx	Rx must points to a ψ-term.
l	*l* is a feature name.
residadr	*residadr* is the next address that will be executed if the constraint residuates. If *residadr* is not present in the instruction, then its value defaults to the subsequent instruction in the program text.
recheckadr	*recheckadr* is the address of the re-evaluation routine. If *recheckadr* is not present its value defaults to the current PC.

The *Feature Existence* instruction tests for the existence of a feature ℓ in the ψ-term in Rx. If the feature exists, then the instruction succeeds and the next instruction is executed. If the feature does not exist, then the instruction residuates and continues execution at *residadr*. The residuation created will restart execution at *recheckadr* if Rx is ever modified.

A possibly more optimized LAM would have a more complex feature existence constraint. Instead of creating a simple residuation which will be run when *any* feature (or sort) of the ψ-term is changed, it could be create a residuation which will only be executed when the particular feature mentioned in the constraint is added the the ψ-term. I have ruled out this more efficient instruction in LAM to keep the residuation structures smaller and more concise. It is not clear whether having the specialized feature residuations would improve efficiency, since whenever a feature is added it would still be necessary to check whether the added feature participates in a residuation.

If LAM is oriented to more efficiency in the case of feature constraints it might, instead, be better to investigate the possibility of actually adding the feature when the constraint is executed. The added feature could point to a ψ-term which has sort top and has a residuation attached that would have to perform a special check related to currying of the original ψ-term. If unification is performed on the original ψ-term no problems are introduced. However, if matching is performed on the ψ-term, some special mechanism would have to be used to ensure that the “ghost” feature added by the constraint was not matched—since, if it were, curried functions would not behave correctly.

I chose the simple route, and instead have feature constraints residuate on the top level ψ-term.

```
    if Rx$\rightarrow$hasFeature(l) then
```

```
        $\text{PC} = \text{PC} + 1$
    **else**
        Rx$\rightarrow$tryAddSimple(CF, *recheckadr*)
        $\text{PC} = \textit{residadr}$
    **endif**
```

$\boxed{\begin{array}{l}\text{Rx} \doteq \text{Ry}? \\ \text{Rx} \doteq \text{Ry}?\ \textit{recheckadr}\end{array}}$ . . . . **Equality Constraint**

Rx	Rx points to a ψ-term.
Ry	Ry points to a ψ-term.
recheckadr	*recheckadr* is the address of the re-evaluation routine. If not present in the instruction it defaults to the current PC.

The *Equality Constraint* instruction checks to see if Rx and Ry are the same dereferenced pointer to a ψ-term. In other words, it succeeds if Rx and Ry have been unified. If they are the same ψ-term, then the constraint succeeds and the next instruction is executed. If they *could* become the same ψ-term, in other words, if they could be unified, then the constraint residuates and the next instruction is executed. If they are inconsistent, then the **headfail** macro is executed. All of the real work performed by this instruction is contained in the canUnify routine (see Section 10.2.3), which will either create a residuation and return or execute the **headfail** macro.

The CF register and CF$\rightarrow$*fail* must be set before this instruction can be executed.

```
    **if** $\text{Rx} \neq \text{Ry}$**then**
        canUnify(Rx, Ry, *recheckadr*)
    **endif**
    $\text{PC} = \text{PC} + 1$
```

$\boxed{\begin{array}{l}X? \\ X?\ \textit{residadr}, \textit{recheckadr}\end{array}}$ . . . . **Initialized Constraint**

residadr	*residadr* is the next address that will <be executed if the constraint residuates. If *residadr* is not present in the instruction, then its value defaults to the current PC+ 2.
recheckadr	*recheckadr* is the address of the re-evaluation routine. If *recheckadr* is not present its value defaults to the current PC.

The *Initialized Constraint* instruction determines whether a memory location points to a ψ-term or is uninitialized. This instruction will either succeed or residuate. It is used to make sure that a variable has been defined before it is used. For instance, if a variable is initialized in part of a constraint tree that could have been skipped (because a feature constraint residuated, i.e., the ψ-term in question was missing a feature), then an equality constraint depending on the skipped variable will have to be skipped until the variable is initialized.

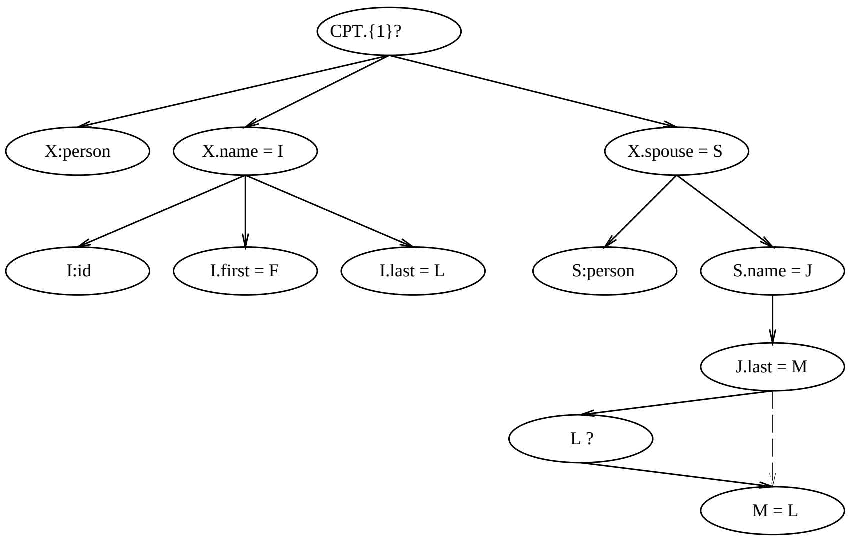


Figure 11: *A constraint tree for the one-argument, one-clause function definition. This tree has been augmented with the Initialized Constraint L?.*

In general, this constraint will be immediately followed by an equality constraint. Since the equality constraint can only be executed if this constraint succeeds, the default address to jump to if the constraint residuates is the one following the next instruction, i.e., the PC+ 2.

For example, the function definition (of one clause with one argument)

$$\begin{aligned} &spouseName(X\!:\!person(name \Rightarrow I\!:\!id(first \Rightarrow F, \\ &\qquad\qquad\qquad\qquad\qquad\qquad last \Rightarrow L), \\ &\qquad\qquad\qquad spouse \Rightarrow S\!:\!person(name \Rightarrow id(last \Rightarrow L)))) \rightarrow body. \end{aligned}$$

creates a constraint tree, as in Figure 11, with an equality constraint between the ψ-terms at the end of the *last* features. Notice that if either the *name* feature of X or the *last* feature of I is not present, then the Equality Constraint instruction $M \doteq L$? in Figure 11 has no meaning. Thus, we introduce the Initialized Constraint instruction L? into the tree in Figure 11 If it residuates, the $M \doteq L$ Equality Constraint will be skipped.

```
**if** $X == 0$ **then**
    $X \mapsto$ tryAddSimple(CF, recheckadr)
    PC $=$ *residadr*
**else**
    PC $=$ PC $+ 1$
**endif**
```

$\boxed{\text{Rx}.\{\ell_1, \ell_2, \cdots, \ell_n\}}$ . . . . **Arity Constraint**

Rx	Rx must point to a ψ-term.

$\ell_1, \ell_2, \cdots, \ell_n$ $\ell_1, \ell_2, \cdots, \ell_n$ is a set of argument names (or features).

The *Arity Constraint* instruction succeeds iff the Rx has the features $\ell_1, \ell_2, \cdots, \ell_n$ and no others. It is used here as a way of specifying the number and names of the arguments to a function. If a function call does not have all the arguments specified in the definition, then the function call becomes curried.

If the Arity Constraint succeeds, then the function call represented by the ψ-term Rx will be executed. If Rx is not a closure (*i.e.*, it has never curried before), then a new ψ-term will be created and assigned to register RR. This ψ-term will become the result of the function. If Rx has previously curried, then a result ψ-term will already have been created, and it is now assigned to RR.

If Rx contains features that are not present in the set $\{\ell_1, \ell_2, \cdots, \ell_n\}$, then the Arity Constraint fails. This should probably also signal some kind of exception, since this kind of failure is not expected. Another implementation of LAM might, instead, pass features not in the set of argument names to the result ψ-term. It is not apparent how such a lenient approach could be implemented efficiently.

If, on the other hand, the features in Rx are a subset of $\{\ell_1, \ell_2, \cdots, \ell_n\}$, then the instruction will curry Rx and return to the caller. A ψ-term is curried by creating a reference link between Rx and a new ψ-term with sort top. A special residuation, called a *handleCurry* residuation, will then be attached to the new ψ-term.

In general the Arity Constraint is the first constraint executed at the head the matching code for a function. It is used to handle the case when the compiler cannot figure out what function will be invoked at compile time and instead must build a complete ψ-term before the function can be invoked. Following the arity constraint will be code that will deconstruct the ψ-term, placing arguments in registers, etc. (see Section 5.1). If, on the other hand, the compiler can determine the function being called, then it can place the arguments into registers (as specified by some mapping of features to registers) and start execution of the function after the Arity Constraint and the deconstruction code.

```
**if** $\mathrm{Rx}\to$features $- \{\ell_1, \ell_2, \cdots, \ell_n\} \neq \emptyset$ **then**
    **headfail**                                  Rx has too many arguments
**else**
    **if** $(\mathrm{Rx}\to$features $\cap \{\ell_1, \ell_2, \cdots, \ell_n\}) == \emptyset$ **then**
        constraint succeeds , setup result $\psi$-term
        **if** $\mathrm{Rx}\to$*ref* $\neq 0$ **then**
            This function had previously curried
            $\mathrm{RR} = \mathrm{Rx}\to$*ref*
            $\mathrm{Rx}\to$*ref* $= 0$
        **else**
            $\mathrm{RR} =$ **new Psi**
        **endif**
        Now execute function body
        $\mathrm{PC} = \mathrm{PC} + 1$
        else
            Must curry this function call
            $C =$ new Psi
            $C \rightarrow$ addCurryResid$($Rx, PC$)$
            ref Rx, $C$                          see the ref instruction
            ret
        endif
    endif
```

resid?
resid? *residadr* . . . . **Residuation Check**

residadr	*residadr* is the next address that will be executed if the frame has any residuations.

The *Residuation Check* instruction checks to see if CF$\rightarrow$*residCount* is zero or not. If it is zero, then the next instruction is executed. If it is not zero then the implicit version, **resid?**,will execute a **ret** instruction and the explicit version, **resid?** *residadr*, will transfer control to *residadr*.

While both the implicit and explicit versions require that the CF register and CF$\rightarrow$*residCount* be initialized, the explicit version gives the user a chance to perform any clean-up that might be required if the function has residuated.

Pseudo-code for the implicit version is

```
    if CF$\rightarrow$residCount $> 0$ then
        ret
    else
        PC $=$ PC $+ 1$
    endif
```

Pseudo-code for the explicit version is

```
    if CF$\rightarrow$residCount $> 0$ then
        PC $=$ residadr
    else
        PC $=$ PC $+ 1$
    endif
```

6.4 Body Instructions

The instructions described in this section can appear in either the head or the body of a function or a predicate definition. These instructions include both constraints and general-purpose instructions.

$\boxed{\mathrm{Rx}{:}s}$ . . . . **Unify with Sort**

Rx	Rx must point to a ψ-term.
s	s is a sort defined in the sort hierarchy.

The instruction Rx:s will succeed if it can set the sort of the ψ-term Rx to $glb(\mathrm{Rx}{\rightarrow}sort, s)$; otherwise it will fail. If the sort of Rx is lowered it will cause any residuations attached to Rx to fire.

```
$s' = \mathrm{glb}(\mathrm{Rx}\rightarrow sort, s)$
**switch** $(s')$
**case** $\perp$ :
    $\mathrm{PC} = \mathrm{CFA}$
**otherwise** :
    $\mathrm{Rx}\rightarrow sort = s'$
    **if** $\mathrm{Rx}\rightarrow rlist$ **then**
        $\mathrm{Rx}\rightarrow \mathrm{lower}()$
    **endif**
**endswitch**
$\mathrm{PC} = \mathrm{PC} + 1$
```

$\boxed{\mathrm{Rx} \leftarrow \mathrm{Ry}.\ell}$ . . . . **Fetch Feature**

Ry	Ry points to a ψ-term.
ℓ	ℓ is a feature name.

The *Fetch Feature* instruction stores the ψ-term at Ry.ℓ in register Rx. It does a destructive store into register Rx. If Ry does not have the feature ℓ, then it will add the feature ℓ to Ry and attach it to a new ψ-term of sort $\top$. The act of adding a new feature will cause any residuations attached to Ry to fire.

```
**if not** $\mathrm{Ry}\rightarrow \mathrm{hasFeature}(\ell)$ **then**
    $\mathrm{Ry}\rightarrow \mathrm{addFeature}(\ell,$ **new Psi**$)$
    **if** $\mathrm{Ry}\rightarrow rlist$ **then**
        $\mathrm{Ry}\rightarrow \mathrm{lower}()$
    **endif**
**endif**
$\mathrm{Rx} = \mathrm{Ry}.\ell$
$\mathrm{PC} = \mathrm{PC} + 1$
```

$\boxed{\textbf{addfeature}\ \mathrm{Rx}, \ell, \mathrm{Ry}}$ . . . . **Feature Creation**

Rx	Rx is a ψ-term without feature ℓ.
ℓ	ℓ is a feature name.
Ry	Ry is the ψ-term that will be pointed to by ℓ.

The *Feature Creation* instruction adds a new feature to ψ-term Rx. This instruction is only valid when it is known that Rx does not have feature ℓ and when Rx has no residuations. In other words, it is used to construct new ψ-terms. After this instruction has been executed, Rx.ℓ? will always succeed, and the sequence Rz $\leftarrow$ Rx.ℓ & Rz $\doteq$ Ry? will always succeed.

```
$\mathrm{Rx} \rightarrow \mathrm{addFeature}(\ell, \mathrm{Ry})$
$\mathrm{PC} = \mathrm{PC} + 1$
```

$\boxed{\begin{array}{l}\mathrm{Rx} \doteq \mathrm{Ry} \\ \textbf{unify}\ \mathrm{Rx}, \mathrm{Ry}\end{array}}$ . . . . **General Unify**

Rx	Rx points to a ψ-term.
Ry	Ry points to a ψ-term.

The *General Unify* instruction unifies the ψ-terms Rx and Ry. If the unification succeeds, then one of Rx and Ry will be modified so that its dereference link points to the other. In general, the younger variable should be modified to point to the older to minimize trailing. If the unification fails then control will be transferred to the address in the CFA register.

```
if unify $(\mathrm{Rx}, \mathrm{Ry})$ then
        $\mathrm{PC} = \mathrm{PC} + 1$
else
        $\mathrm{PC} = \mathrm{CFA}$
endif
```

$\boxed{\textbf{ref}\ \mathrm{Rx}, \mathrm{Ry}}$ . . . . **Create Reference Link**

Rx	Rx points to a ψ-term.
Ry	Ry points to a ψ-term.

The *Create Reference Link* instruction sets the reference field of the Rx ψ-term to point to Ry. After this instruction has executed Rx $\doteq$ Ry? would succeed.

```
$\mathrm{Rx} \rightarrow \mathit{ref} = \mathrm{Ry}$
$\mathrm{PC} = \mathrm{PC} + 1$
```

$\boxed{\textbf{deref}\ \mathrm{Rx}, \mathrm{Ry}}$ . . . . **Retrieve a Dereferenced ψ-term**

Rx	Rx points to the ψ-term to be dereferenced.
Ry	Ry the dereferenced ψ-term is returned here.

This instruction will trace down the reference links starting with the ψ-term in Rx. It will put the ψ-term at the end of the chain into Ry. As a shorthand, "**deref** Rx, Ry" will often be written "Ry= Rx". In other words, the code presented here will not draw a clear distinction between the case where an assignment is sufficient and the case where a **deref** is necessary. The compiler can make all assignments **deref** s, or, if a good optimizer is available, can eliminate some **deref** operations in favor of simpler moves.

```
$\mathrm{Ry} = \mathrm{Rx}$
while $\mathrm{Ry}\rightarrow ref \neq 0$
        $\mathrm{Ry} = \mathrm{Ry}\rightarrow ref$
endwhile
$\mathrm{PC} = \mathrm{PC} + 1$
```

$\boxed{\begin{array}{l}\textbf{eval}\ \mathrm{Rx}\\ \textbf{evalfunc}\ \mathrm{Rx}\\ \textbf{evalpsi}\ \mathrm{Rx}\end{array}}$ . . . . **Evaluate by Normalization**

Rx	Rx points to a ψ-term.

The evaluation instructions cause a ψ-term to be made consistent. If the ψ-term is a function call or a closure, then the evaluation instruction will attempt to evaluate the ψ-term by executing the function based on the ψ-term's sort. If the sort of the ψ-term is not a function, then the evaluation instruction will enforce the sort definition on the ψ-term. If there is no sort definition, then the instruction is a nop.

This instruction is a full fledged interpreter and as such is very costly. It is hoped that after compilation there are few instances of this instruction and that instead, the unification, matching, and normalization that it implies has been encoded with by the other instructions presented here.

The general **eval** instruction makes no assumption about whether Rx is a function or a sort. **evalfunc** assumes that Rx points to a ψ-term with a function sort. **evalpsi** assumes that Rx points to a normal ψ-term.

```
$adr = \mathrm{lookupSort}(\mathrm{Rx}\rightarrow sort)$
call $adr$
```

new Psi
new Psi(s)
new Frame$(name)$
new SortResid$(recheckadr, \mathrm{Rx}, \mathrm{Ry})$
. . . . **Heap Allocation**

s	*s* is a sort.
name	*name* is the name of a function.
recheckadr	*recheckadr* is the address of the re-evaluation routine.
Ry	Ry is the ψ-term that residuated.

The *Heap Allocation* instructions allocate storage from the heap. **new Psi**(s) will allocate a new ψ-term with sort *s* and no features. If *s* is not present it defaults to top. **new Frame**(name) allocates a new frame for the function *name*. It will also set every slot in the frame to 0. It is assumed that the compilation process will create a routine for allocating a new function frame for each function. It is this routine that will be invoked when the **new Frame**() routine is executed. A more general way of looking at this would be to say that for every sort a routine is created that will handle its allocation and initialization. For general sorts this will allocate some space and set the sort to top. (One could imagine that if a sort definition were present for the sort then this routine would also execute the code in the definition.) For sorts that represent functions, this code would allocate the frame and initialize it.

new SortResid(recheckadr, Rx, Ry) allocates a new sort residuation which will resume at *recheckadr*. The residuation will be for the frame pointed to in Rx, and it will be attached to the ψ-term Ry.

6.5 Obvious instructions

The following instructions are used in the examples and should be obvious.

free Rx . . . . **Free Heap Allocated Storage**

push Rx . . . . **Push onto the Stack**

$[\mathrm{SP}] = \mathrm{Rx}$
$\mathrm{SP} = \mathrm{SP} + 1$

pop Rx . . . . **Pop from the Stack**

$\mathrm{SP} = \mathrm{SP} - 1$
$\mathrm{Rx} = [\mathrm{SP}]$

ret . . . . **Return from Subroutine**

$\mathrm{SP} = \mathrm{SP} - 1$
$\mathrm{PC} = [\mathrm{SP}]$

$\boxed{\textbf{call}\ \textit{destadr}}$ . . . . **Call Subroutine**

$[\text{SP}] = \text{PC}$
$\text{SP} = \text{SP} + 1$
$\text{PC} = \textit{destadr}$

$\boxed{\textbf{jmp}\ \textit{destadr}}$ . . . . **Unconditional Branch**

$\text{PC} = \textit{destadr}$

7 Example LAM Code

This section illustrates how LIFE functions are compiled for the LAM. It uses append as an example function. First it presents a simple, ψ-term based function definition. Next it optimizes the code to use registers for the recursive call. Finally it explores how the explicit versions of certain instructions can substantially reduce overhead. We will first explore an example using the append function in Section 8. Then in Section 9 we will show how the mechanism of resumption works.

8 The Append Function

```
append([], L) → L.
append([H|T], L) → [ H | append(T, L) ].
```

In general an n-rule function in LIFE is compiled into $2n + 2$ sections. The first section is the ψ-term entrance to the function. It consists of an arity instruction followed by the instructions to deconstruct the ψ-term into registers. The next section is the frame building section. It allocates a frame and stores necessary information into the frame. Finally, for each rule a head section is followed by a body section.

In the following non-optimized version of `append` the first section starts at the label append, which contains the arity constraint and the deconstruction instructions. The frame building section begins at the label reg_append. The first clause's head is only two instruction long. They are right before the label isNil. The body of the first clause is the 5 instructions following isNil. The second clause's head starts at maybeList and its body starts at isList. In this version of `append` no special attempt is made to use registers; instead it builds a ψ-term for the recursive call to itself.

The frame definition for append is as follows.

```
class Append : public Frame
{
   Psiterm*    arg1;           // pointer to argument 1
   Psiterm*    L;              // pointer to argument 2
};
```

append:	CPT.$\{1,2\}$	Check that caller has two arguments. This is the ψ-term entrance section.
	R1 $\leftarrow$ CPT.1	
	R2 $\leftarrow$ CPT.2	
	ref CPT, RR	Any reference to the function will now point to the result of the function.
reg_append:	**push** CF	This is the register interface entry. Allocate a new frame.
	CF $=$ **new Frame**$($*append*$)$	
	CF$\rightarrow$*body* $=$ isNil	Setup for executing the first rule.
	CF$\rightarrow$*fail* $=$ maybeList	If the first rule fails, try the second one.
	CF$\rightarrow$*result* $=$ RR	
	CF$\rightarrow$*arg*1 $=$ R1	
	CF$\rightarrow$*L* $=$ R2	
	R1$\subseteq$[ ] zero	Is the first rule ok?
	resid?	
isNil:	CF$\rightarrow$*fail* $= 0$	Any failure now is a general failure.
	unify CF$\rightarrow$*result*, CF$\rightarrow$*L*	Since we don't know if there is already a result, we must assume there is and do a general unification.
	free CF	Clean up and return.
	pop CF	
	ret	
zero:	CPT$\subseteq$[ ]	Recheck sort constraint from rule 1.
	ret	
one:	CPT$\subseteq$*List*	Recheck subsort constraint from rule 2.
	ret	
two:	CPT.*car*?	Recheck existence constraint.
	ret	
three:	CPT.*cdr*?	Recheck existence constraint.
	ret	
maybeList:	CF$\rightarrow$*body* $=$ isList	First rule failed, so setup for executing second rule.
	CF$\rightarrow$*fail* $= 0$	If this rule fails, the whole function call fails.
	R1 $=$ CF$\rightarrow$*arg*1	Begin checking head of second rule.
	R1$\subseteq$*List* one	
	R1.*car*? $\$+1$, two	
	R1.*cdr*? $\$+1$, three	

```
          resid?

isList:   CPT = new Psi(append)              Executing second rule, so call append
                                             recursively.
          addfeature  CPT, 1, (CF→arg1).cdr
          addfeature  CPT, 2, CF→L
          evalfunc  CPT                      Perform actual function call.

          R2 = new Psi(List)                 Build cons cell.
          addfeature  R2, car, (CF→arg1).car
          addfeature  R2, cdr, RR
          evalpsi  R2

          unify  CF→result, R2               Make it the result.

          free  CF                           Clean up and return.
          pop  CF
          ret
```

We can optimize append in two ways. First, compiler analysis should let us check the function definition to see if the function call we are performing satisfies the arity constraint. If it does, we can eliminate building the ψ-term. In the case of append, the recursive call in the second clause can use registers instead of constructing a ψ-term. Further, compiler analysis can also check sort definitions to see if a ψ-term being built will not violate its theory. (In the trivial case this happens when the sort has no attached theory.) For append, this results in the following optimizations applied to the second clause body:

```
isList:   R1 = (CF→arg1).cdr                 R1 will get first arg to append.
          R2 = CF→L
          RR = new Psi                       Caller must establish the result ψ-term.
          call  reg_append                   Execute recursive call and then the rest
                                             is the same.

          R2 = new Psi(List)
          addfeature  R2, car, (CF→arg1).car
          addfeature  R2, cdr, RR

          unify  CF→result, R2

          free  CF
          pop  CF
          ret
```

The above versions of append use only the implicit versions of the instructions. While the implicit versions are more compact they also require more setup, which means that for the simple case here more instructions are executed to establish a context than there are instructions in the head or body of the function. Another defect with this code is that the common case, the second rule, is the slowest case. To alleviate this problem, the explicit subsort and residuation check instructions can be used.

Since the explicit instructions make no assumptions about what is in the current frame, nothing in the frame needs to be set up before executing the head of the function. Thus, if the function does not residuate, no extra work will be performed in setting up the body, fail, or result fields of the frame. Notice also that more registers are used as a result of this compilation approach. The extra overhead involved when a function residuates is smaller than it appears here because on a RISC processor everything would have to be loaded into registers anyway. Furthermore, some of the work that is made explicit here was actually happening behind the scenes in the definitions of the implicit instructions.

append:	CPT.$\{1, 2\}$	ψ-term entrance to function.
	R1 $\leftarrow$ CPT.1	
	R2 $\leftarrow$ CPT.2	
	ref CPT, RR	
reg_append:	**push** CF	Save current value of CF.
	CF $=$ **new Frame**$($*append*$)$	Create a new frame. All values are set to 0 by allocator.
	R1$\subseteq$[ ] maybeList, residNil	Explicit subsort constraint will only continue to next instruction if it succeeds. So no **resid?** is needed.
isNil1:	**unify** RR, R2	We can assume everything needed is in a register and CF$\rightarrow$*fail* is already set to 0.
	free CF	
	pop CF	
	ret	
isNil:	RR $=$ CF$\rightarrow$*result*	Entry point for body if came back from a residuation. Must setup registers.
	R2 $=$ CF$\rightarrow$$L$	
	jmp isNil1	
checkNil:	CPT$\subseteq$[ ]	Recheck sort constraint from rule 1.
	ret	
residList:	CF$\rightarrow$*body* $=$ isList1	If rule 2 residuates on the first pass we end here to fixup the frame.

	$\mathrm{CF}\rightarrow \mathit{fail} = 0$	
	jmp stash	
residNil:	$\mathrm{CF}\rightarrow \mathit{fail} = \mathrm{maybeList1}$	If rule 1 residuates on the first pass we end up here to fixup the frame and to create the residuation. Since rule 1 uses an explicit form of the subsort instruction we must create the residuation ourselves.
	$\mathrm{CF}\rightarrow \mathit{body} = \mathrm{isNil}$	
	$\mathrm{CF}\rightarrow \mathit{residCount}{+}{+}$	
	$\mathrm{R1}\rightarrow \mathit{tryAddSimple}(\mathrm{CF}, [\,], \mathit{checkNil})$	
stash:	$\mathrm{CF}\rightarrow \mathit{result} = \mathrm{RR}$	
	$\mathrm{CF}\rightarrow 1 = \mathrm{R1}$	
	$\mathrm{CF}\rightarrow L = \mathrm{R2}$	
	ret	
one:	$\mathrm{CPT}\subseteq \mathit{List}$	Recheck subsort constraint for rule 2.
	ret	
two:	$\mathrm{CPT}.\mathit{car}?$	
	ret	
three:	$\mathrm{CPT}.\mathit{cdr}?$	
	ret	
maybeList1:	$\mathrm{R1} = \mathrm{CF}\rightarrow 1$	Entry point if rule 1 resumes after residuating and then fails.
	$\mathrm{R2} = \mathrm{CF}\rightarrow L$	
	$\mathrm{CF}\rightarrow \mathit{fail} = 0$	
	$\mathrm{CF}\rightarrow \mathit{body} = \mathrm{isList1}$	
maybeList:	$\mathrm{R1}\subseteq \mathit{List}$ one	Entry point if rule 1 fails on the first pass. Everything needed is in a register and the current frame may be uninitialized.
	$\mathrm{R1}.\mathit{car}?$ $\$+1$, two	
	$\mathrm{R1}.\mathit{cdr}?$ $\$+1$, three	
	resid? residList	Since the frame may be uninitialized, we must do some work before returning to caller if we are to residuate.
	$\mathrm{CF}\rightarrow 1 = \mathrm{R1}$	We did not residuate, but we may never have stored anything in the frame, so save what we need to.
	$\mathrm{CF}\rightarrow \mathit{result} = \mathrm{RR}$	
isList:	$\mathrm{R1} = \mathrm{R1}.\mathit{cdr}$	
	RR = **new Psi**	
	call reg_append	

```
            R2 = **new Psi**(List)
            **addfeature** R2, car, (CF→1).car
            **addfeature** R2, cdr, RR

            **unify** CF→result, R2

            **free** CF
            **pop** CF
            **ret**

isList1:    R1 = CF→1                    Entry point if we succeed after residu-
                                         ating.
            R2 = CF→L
            **jmp** isList
```

This version of append is significantly harder to read than the previous versions for two reasons. First, the code is organized so that the most common case requires the fewest jumps. Thus, all the interesting (but hopefully rare) cases involve many jumps. Second, one must keep in mind the different states that the frame can be in to determine which code is executed. In general the compiler will need to provide code for two different states: initial entry and resumed entry.

For instance, in this code the second clause can be entered in one of three ways: failure of the first clause in the first pass, failure of the first clause after it residuated, or residuation of the second clause. In the first case, the flow of control passes though reg_append and then to maybeList via the explicit subsort constraint. In this case none of the frame has been established. In the second case, the flow of control passes through reg_append, residNil (because the explicit subsort residuated), checkNil (when the frame was resumed), maybeList1 (because the constraint in checkNil failed and the fail of the field was set to maybeList1 in residNil). In the second case the frame has already been initialized. In the third case the frame also has been initialized. The flow of control is: reg_append, maybeList, residList, one of the resumed labels (one, two, or three), and then, finally, isList1.

9 The Plus Function

In order to clarify the mechanisms used for residuation and resumption we will examine the flow of control for the following contrived segment of LIFE code.

```
A=@, B=@, C=A+B, D=C+5, A=4, B=5
```

We will assume that + is not a built-in function, but rather is defined by the one clause LIFE code:

```
+(A:int, B:int) -> intplus(A, B).
```

Where `intplus(A, B)` treats the sorts of `A` and `B` as integers and returns a a ψ-term with the sort that is an integer–that is their sum.

The frame definition and compiled LAM code for this function are as follows.

```
class Plus : public Frame
{
   Psiterm*    arg1;
   Psiterm*    arg2;
};
```

plus:	CPT.$\{1, 2\}$	Check that caller has two arguments. This is the ψ-term entrance section.
	R1 $\leftarrow$ CPT.1	
	R2 $\leftarrow$ CPT.2	
	ref CPT, RR	Any reference to the function will now point to the result of the function.
reg_plus:	R1$\subseteq$*int* fail, residFirst	Explicit subsort constraint will execute general fail routine if this constraint fails, since this definition has only one clause.
	R2$\subseteq$*int* fail, residSecond	
body:	R3 $=$ *intPlus*(R1, R2)	Execute the add primitive and put the result, a ψ-term, into R3
	unify RR, R3	Unify the resulting number with the result ψ-term. Compiler analysis should do better here, since we know intPlus returns a ψ-term that has no features and RR is a null ψ-term.
	ret	
residFirst:	**push** CF	Save current value of CF.
	CF $=$ **new Frame**(*append*)	
	R1$\rightarrow$*tryAddSimple*(CF, *int*, *checkPlus*)	
	R2$\subseteq$*int*	Since the frame is now all set and we know we are residuating already, use an implicit subsort constraint for second argument.
stash:	CF$\rightarrow$*body* $=$ doPlus	
	CF$\rightarrow$*residCount*++	
	CF$\rightarrow$*one* $=$ R1	
	CF$\rightarrow$*one* $=$ R2	
	pop CF	
	ret	
residSecond:	**push** CF	
	CF $=$ **new Frame**(*append*)	
	R2$\rightarrow$*tryAddSimple*(CF, *int*, *checkPlus*)	
	jmp stash	

checkPlus:	$\text{CPT}\subseteq \mathit{int}$	Recheck sort constraint
	ret	
doPlus:	$\text{R1} = \text{CF}\rightarrow \mathit{one}$	
	$\text{R2} = \text{CF}\rightarrow \mathit{two}$	
	$\text{RR} = \text{CF}\rightarrow \mathit{result}$	
	$\text{R3} = \mathit{intPlus}(\text{R1}, \text{R2})$	Execute the add primitive and put the result, a ψ-term, into R3
	unify RR, R3	Unify the resulting number with the result ψ-term.
	pop CF	
	ret	

The above function has two additional places for optimizations than the append function. First, both arguments have the same sort constraint, so only one resumption address is needed, *checkPlus*. Second, in the case that no residuations occur, which is the common case, no frame needs to be created either. Thus, we see that the CF register is not pushed nor is a frame created unless a residuation occurs (see labels *residFirst* or *residSecond*).

Armed with the above definition for plus, lets see what happens for our example LIFE code.

```
A=@, B=@, C=A+B, D=C+5, A=4, B=5
```

The compiled version of this code is as follows (We assume that none of these variables has been seen before.).

start:	$A =$ **new Psi**$(@)$	Create a new psi term for A
	$B =$ **new Psi**$(@)$	Create a new psi term for A
	$\text{R1} = A$	Get ready for a function call
	$\text{R2} = B$	
	$\text{RR} =$ **new Psi**$(@)$	Create the result term
	call reg_plus	
	$C = \text{RR}$	
afterOne:	R1 = '5'	Put sort for the integer 5 in R1 for second call
	$\text{R2} = \text{RR}$	
	$\text{RR} =$ **new Psi**$(@)$	Create the result term
	call reg_plus	
afterTwo:	$D = \text{RR}$	
lowerA:	*A*:'4'	Unify A with the integer 4
lowerB:	*B*:'5'	Unify B with the integer 4

After the execution of the first call, from the label *start* until the instruction before label *afterOne*, three ψ-terms, one frame, and two residuations will have been created. This is pictured in Figure 12. When the second call finishes (right before the label *afterTwo*, another

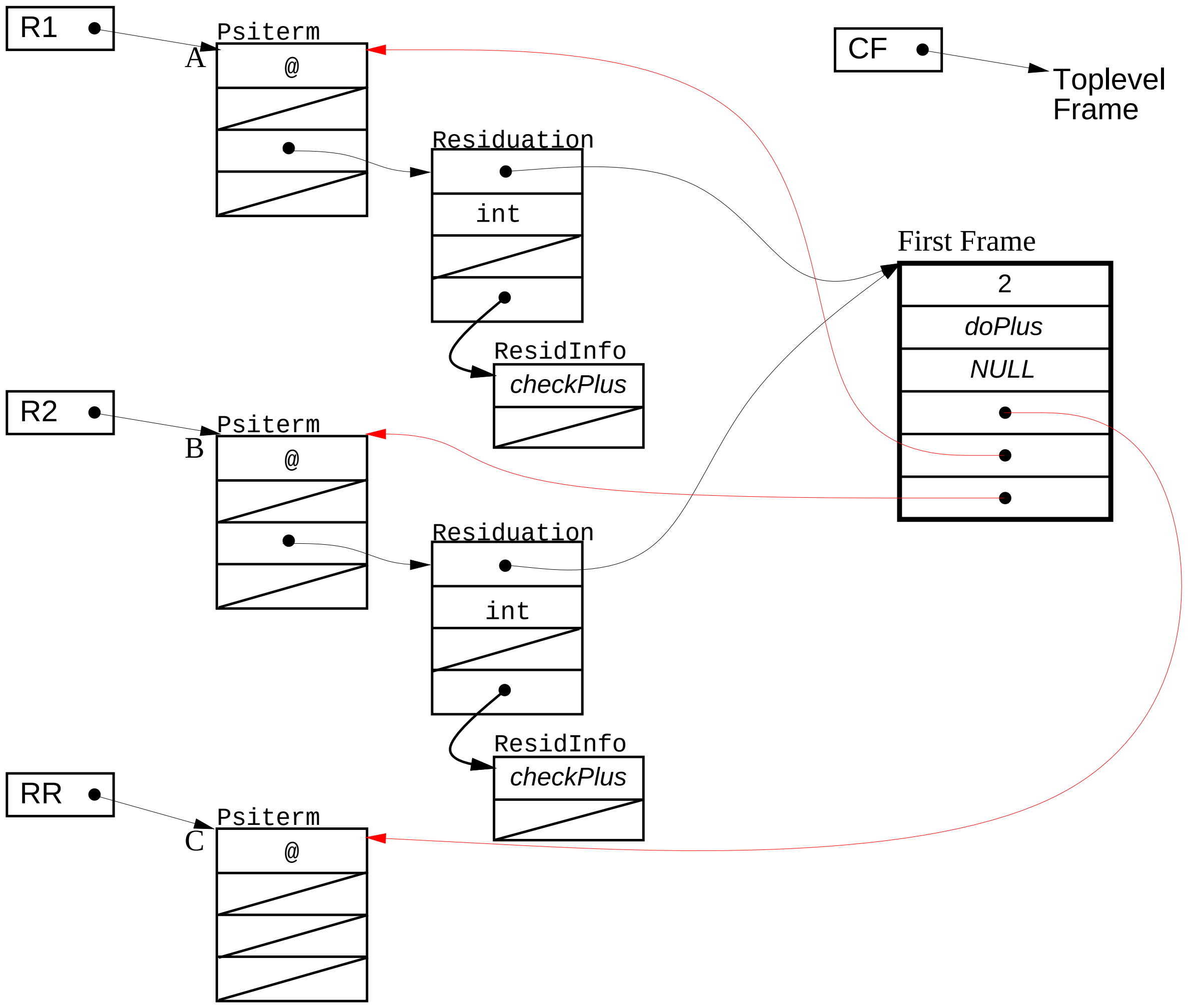


Figure 12: *The structures created by executing the first seven instructions of the example code.*

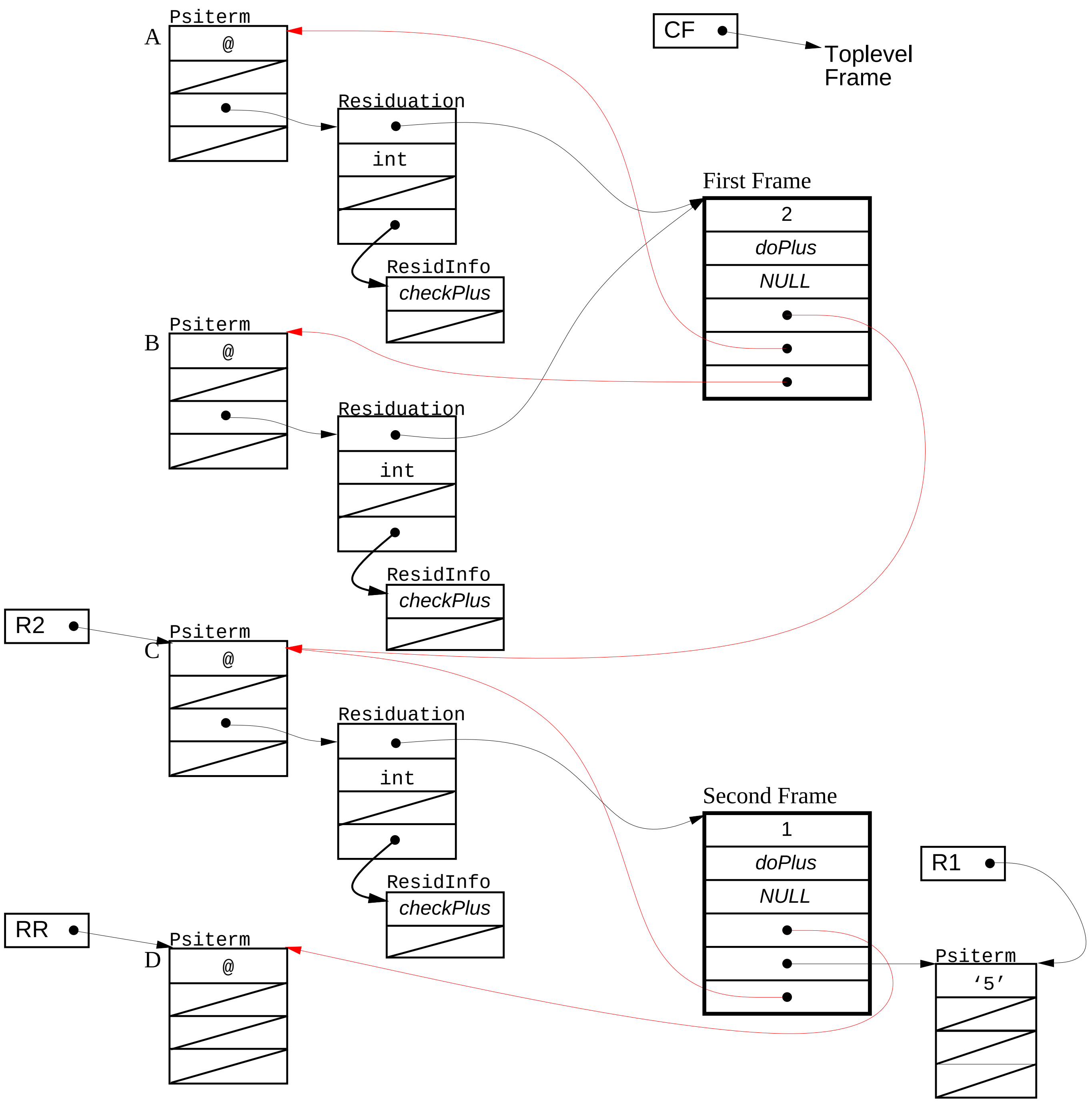


Figure 13: *The structures created after the second call to the plus.*

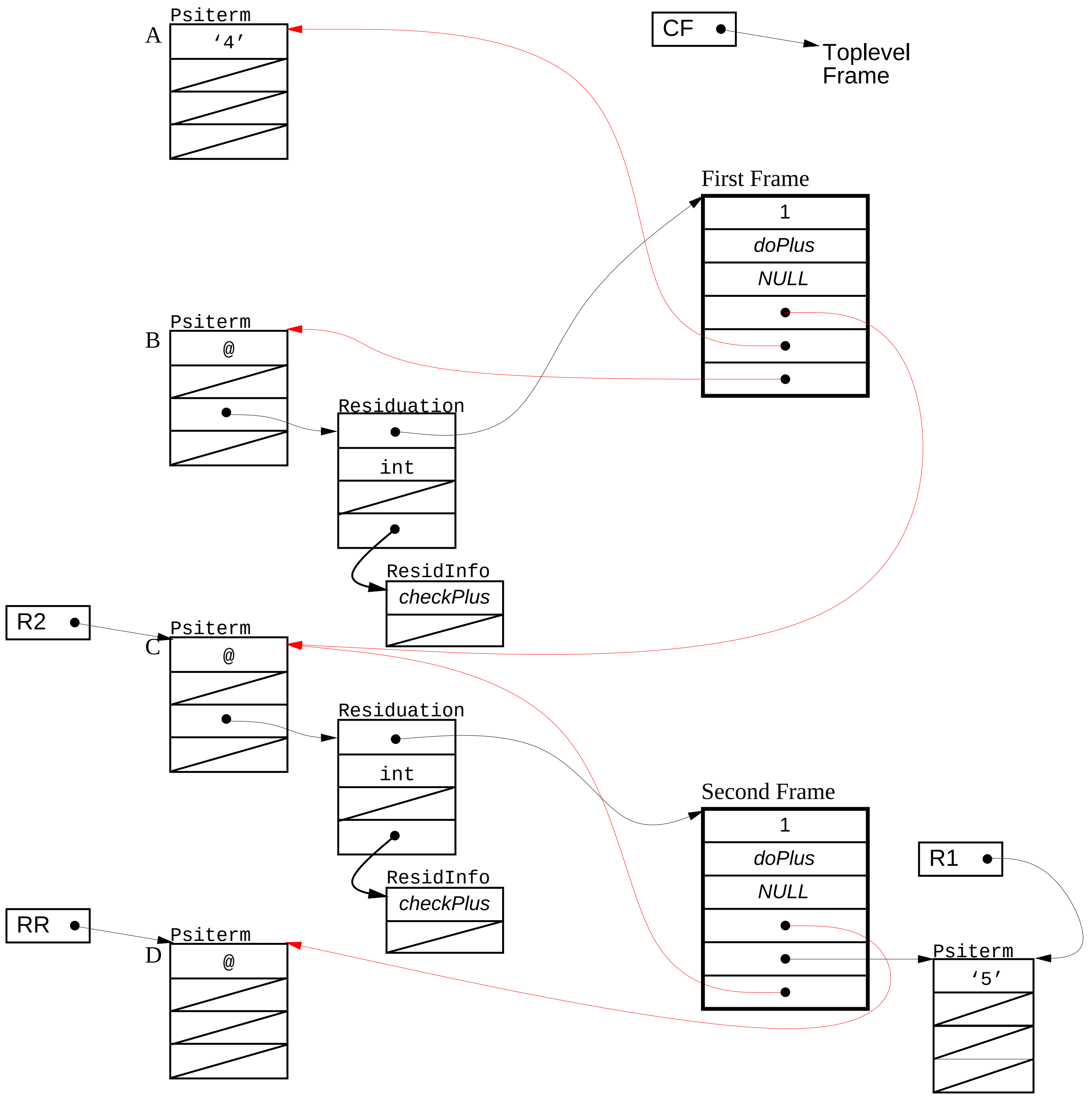


Figure 14: *After A has been lowered and its residuation removed.*

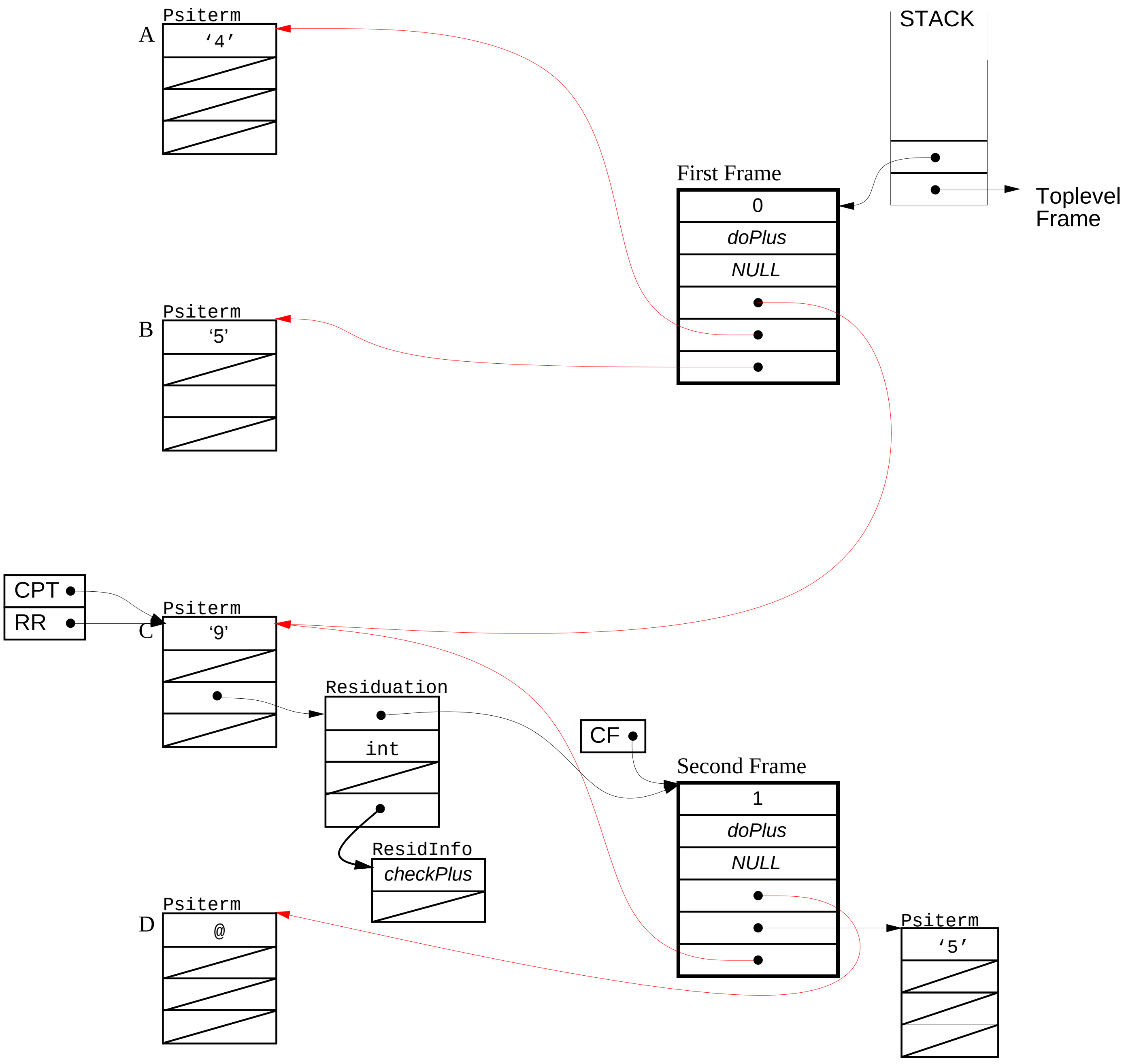


Figure 15: *The structures and stack after the first frame has fired, but before the second one has fired.*

ψ-term, frame, and residuation will be created (see Figure 13). Note that in executing the second call, the first subsort constraint succeeds, but the second one (testing the sort of *C*) residuates. At this point, both result terms have sort top, and neither of the functions have fired. Next, the ψ-term *A* will be lowered to the integer 4. This causes the residuation attached to *A* to fire. The result is that the current Frame is pushed on the stack, the CPT register is set to *A*, and the code at *checkPlus* is executed. After this returns, the residuation attached to A will be removed and the the residCount attached to the first frame will be lowered to one (See Figure 14).

Finally, B is lowered. This will cause *checkPlus* to be executed with CF pointing to the first frame. All the residuations will have been removed and thus, the `execute()` function (See Page 52) will invoke the first frame.

When the first frame executes, it will unify the sorts between RR, which is top, and R3 which contains a '9'. The result is a lowering of RR, which is also the ψ-term *D*, causing another residuation to fire. Again the CF register is pushed, *checkPlus* executed (See Figure 15 for the state of the machine just before *checkPlus* is called), and the *intPlus* routine finally called, putting a 14 in *D* as expected. The second frame returns, popping the CF for the first frame. The first frame returns, popping the top level CF, and then the code continues.

This example is contrived and given the above code a peephole optimizer should be able to rearrange this so no residuations occur.

10 Detailed Data Structures and Routines

This section describes the structure of the basic constraints needed to execute matching. It defines the data structures, auxiliary routines, and abstract machine structure necessary to understand the execution of the constraints. The definitions are presented in an object-oriented fashion. It is hoped that the definitions presented here can be turned into executable C++ code.

10.1 The Three Basic Data Structures

10.1.1 The Psiterm

```
// This class defines the basic data structure used in the system, that of the psiterm.
// Each psiterm has a sort, a set of features, and a potentially null list of residuations
// attached to it.

class Psiterm
{
    Sort            sort;
    Features        features;
    Residuation     rlist;
public:

    // return sort of this psiterm
    Sort            sort();
    // Return residuation for frame f; if it doesn't exist create it.
    Residuation*    findResid(Frame *f);

    // If s is compatible with best known sort for this psiterm, then add a simple
    // residuation that will continue at label for frame f.
    boolean         tryAddSimple(Frame *f, Sort s, Code label);

    // Return a the list of features that is the intersection between fset and this
    // psiterms features list.
    Features        intersect(Features fset);

    // Add feature l with attached psi-term p.
    void            addFeature(Feature l, Psiterm* p);

    // return true if this psiterm has the feature l.
    boolean         hasFeature(Feature l);

    // get the psiterm that is pointed to by feature f.
    Psiterm&        get(Feature f);

    // this Psiterm has been lowered, so if it has any residuations, they must be
    // executed.
    void            lower();

    // Add a residuation that will invoke the handleCurry routine if anything is ever
    // unified to this psiterm. The original function call psiterm is specified by orig.
    // The function to be invoked is func.
    void            addCurryResid(Psiterm& orig, Code func);

    // return true if X and Y are unifiable. Also set up the structures that will ensure
    // they remain compatible or a failure takes place.
    friend boolean  canUnify(Psiterm& X, Psiterm& Y);
};
```

10.1.2 The Frame

The name for this class was chosen to evoke the image of a stack frame used to execute a function. However, since residuations can cause functions to execute in a non-stacklike manner, they cannot be allocated on a sequential stack. Everything that the function references across suspension, or residuation points, is kept in the frame. The class frame describes the common structure that every frame must have. Each function will create a new subclass of frame, which will add all the variables needed in order to execute the function.

```
// The class Frame describes the minimum structure that every function will create when
// it is executed. Each particular function will define its own additional fields (i.e.,
// variables needed to execute the function) and inherit all of the class Frame's
// variables and methods.

class Frame
{
    int             residCounter;    // counts number of residuated variables.
    Code            body;            // address of body of function
    Code            fail;            // the address to goto if a fail is executed.
    Psiterm*        result;          // result of function gets put here

public:
    // increment and decrement the residuation counter
    void            incrResiduation();
    void            decrResiduation();

    // returns residCounter to indicate whether or not this frame has any residuations
    // attached to it.
    void            hasResids();

    // when this function is called, then all residuations have been checked and the
    // function is ready to go.
    void            execute();
};

// An example frame definition. It inherits from Frame and defines two of its own
// variables.

class SomeFunction : class Frame
{
    Psiterm*        V0;
    Psiterm*        V1;
}
```

10.1.3 The Residuation

```
// A residuation is a structure containing all the information about functions that
// depend on psiterms on which they are residuated. A residuation identifies the function
// depending on the psiterm, the best known sort for the psiterm (i.e., the glb of all the
// formals in the function definition that this psiterm is participating in), and a list
// of addresses that should be executed if the psiterm is ever changed.

class Residuation
{
    Frame*          parent;      // frame of function to be activated
    Sort            sort;        // the best known sort for this term (i.e. the glb of
                                 // all formals and all actuals linked by equality
                                 // contraints)
    Residuation*    next;        // next residuation for this var
    ResidInfo*      info;        // info about each resid for this parent

public:
    // Compare the sort in this residuation and s. If they are incompatible, then
    // execute FAIL. Otherwise, set sort to glb(sort, s).
    boolean         compatibleWith(Sort s);
    // create a ResidInfo and attach it to the list of already created residInfo's for
    // this psiterm.
    void            addSimpleResid(Code label);

    // return best known sort, i.e. return sort field.
    Sort            sort();

    // set the sort field to s. If s is different than the current value of the sort,
    // then the best known sort for this psiterm has been changed and we should pretend
    // the psiterm has been lowered.
    void            setSort(sort s);

    // returns true if y has an equality residuation in frame f with the receiver;
    // otherwise it returns false.
    boolean         hasEqResidWith(Psiterm& y, Frame *f);

    // adds an equality residuation between the this psiterm and x.
    void            addEqResidWith(Psiterm& x, Code label);

    // Indicates that the psiterm connected that this residuation is attached to has
    // been lowered, so we should activate this residuation.
    void            lower();

    // Add a ResidInfo to this residuation that should resume at label.
    void            addSimpleResid(Code label);
};

// these two classes hold the address of the code chunk to be executed if the psiterm
// pointing to these is ever lowered.

class ResidInfo
{
    Code            address;          // address of constraint to re-execute
    ResidInfo*      next;             // next ResidInfo for this frame if it exists, else
                                      // NULL

  public:
    // This function calls the routine at address. CPT and CF have already been set up.
    virtual void    resume();
};

class EqResid: ResidInfo
{
    Psiterm         other;            // psiterm used in = constraint

  public:
    // This function executes the equality constraint between the psiterm in CPT and the
    // Psiterm in other. If it succeeds, then the routine at address is executed.
    virtual void    resume();
};
```

10.1.4 The Sort

```
// The class sort represents the sorts in the system. The choice of representation
// should be made with efficiency of taking the glb in mind.

class Sort
{
  public:
    // Return TRUE is this is bottom sort, otherwise FALSE.
    Boolean isBottom(void);

    // return the greatest lower bound of the sorts defined by a and b.
    friend Sort glb(Sort& a, Sort& b);
};
```

10.2 The Head Instructions

In this section the C++ code for the basic head instructions listed in Section 6.3 is presented. Each constraint is listed using the data structures and routines presented above. In some sense this code represents the macros that would be inlined into a LAM function definition. The code uses three macros to guide the flow of control in the abstract machine: `CONTINUE`, `FAIL`, and `SKIP_TO`. The `CONTINUE` macro has essentially the same meaning as $\text{PC} = \text{PC} + 1$ in the pseudo-code presented in Section 6.3. In these macros it means execute the code after the end of the current macro. The `FAIL` macro is like the **headfail** macro. Finally, `SKIP_TO(skip)` means to restart execution at the instruction labeled `skip`. These macros imply a certain compilation regime, that the entire function head will be encapsulated in a single C function. Thus, both the `SKIP_TO` and `CONTINUE` macros, become local gotos, and the `FAIL` macro becomes a return statement. If the function returns the address of the next c function to execute, then on failure a fail routines address can be returned, on success, the next LIFE function routine's address can be returned.

10.2.1 Sort Constraint

See Page 17 for the pseudo-code definition of this constraint.

```
void
subsort(Psiterm& r,                // psiterm that we are testing
        Sort s,                    // the sort it must agree with
        Code label)                // address of code to resume if this residuates
{
    Sort           g;

    g = glb(r.sort(), s);
    if (g.isBottom()) FAIL;        // if it is bottom, then FAIL
    if (g <= s) CONTINUE;          // if it is under s, succeed.
    // we must residuate. so create a residuation for the current frame which will have
    // a sort compatible with g and will continue at 'label' when the psiterm is lowered.
    // If tryAddSimple can't create the residuation (because g is incompatible with the
    // sort already in this psiterm's residuation for the current frame) then it will
    // execute the FAIL macro.

    r.tryAddSimple(CF, g, label);

    CONTINUE;
}
```

10.2.2 Feature Constraint

See Page 19 for the pseudo-code definition of this constraint.

```
void
featureExistence(Psiterm &r,     // The psiterm that we are testing.
                 Feature l,      // The feature we want to check for.
                 Code label,     // The address to resume to.
                 Code skip)      // The place to continue if this constraint residuates.
{
    // Does r have the feature l? If so succeed.

    if (r.hasFeature(l)) CONTINUE;

    // OK, guess we have to residuate

    r.tryAddSimple(CF, g, label);

    // Continue execution at 'skip'

    SKIP_TO(skip);
}
```

10.2.3 Equality Constraint

See Page 20 for the pseudo-code definition of this constraint.

```
void
equality(Psiterm& r0,            // The psiterm we check to be equal with r1.
         Psiterm& r1,            // The psiterm we check to be equal with r0.
         Code label)             // The address to continue at on resumption.
{
    // If r0 and r1 both point to same psiterm, we're golden.

    if (r0 == r1) CONTINUE;

    // All the real work happens in

    canUnify(r0, r1);
    // If we get here then the psiterms can be unified, so

    CONTINUE;
}

// The heart of the equality constraint is the canUnify routine. It checks to see that
// two psiterms can be unified. This is an expensive procedure that not only checks the
// sorts of its two arguments, but all features of each term that could be unified.

void
canUnify(Psiterm& x, Psiterm& y)
{
    Residuation* rx;                // The x's residuation for this frame.
    Residuation* ry;                // The y's residuation for this frame.

    rx = x.findResid(CF);           // Get (and if necessary create) x's residuation for the
                                    // current frame.

    if (rx->hasEqResidWith(y, CF))
    {
        // If x and y are already involved in an equality constraint, then we have
        // already checked to see that they (and all the psiterms reached through their
        // features) are compatible, so we just return. This is in essence the mark that
        // is used to stop infinite loops from happening in this canUnify routine.

        return;
    }

    // x and y haven't been checked yet, so create a resid for y

    ry = y.findResid(CF);

    // Get the glb of the BEST KNOWN sorts of x and y.

    g = glb(rx->sort(), ry->sort());

    if (g.isBottom()) FAIL;

    // Force both x's and y's best known sort to be their glb. The setsort procedure
    // will also invoke any residuations that might already be attached to x and y.

    rx->setsort(g);
    ry->setsort(g);

    // Add an equality residuation to each of rx and ry.

    rx->addEqResidWith(y, label);
    ry->addEqResidWith(x, label);

    // Now each of the features that are in common between x and y must be checked to
    // see if they can unify.

    foreach (f, x.intersect(y.features()))
    {
        canUnify(x.get(f), y.get(f));
    }

    // if we made it this far then x and y can unify
}
```

The complexity of the equality constraint and in particular the `canUnify` procedure described above is due to catching disentailment caused by argument unification. To see how the above code works we will work out an example based on the sort hierarchy in Figure 4, the function definition

```
theSame(X, X, X) → 1.
```

and the call

```
theSame(@(x ⇒ a), b(x ⇒ b, y ⇒ b), c(x ⇒ c))?
```

First note that the LAM code produced for the function definition will include the following segment:

$$R1 \doteq R2?$$
$$R2 \doteq R3?$$

We assume that the registers have been loaded with the first, second and third arguments respectively. Upon execution of the first constraint the ψ-terms and their associated residuations will be as shown in Figure 16. First, note that the `bestsort` field of the residuations for the ψ-term in R1 is set to `b`. Second, note that there are no residuations attached to the ψ-term attached to R1$\rightarrow$*y*.

When the second constraint is executed, the `canUnify` routine will be invoked on the ψ-terms pointed to by registers two and three. `rx` will point to the residuation for R2 in Figure 16. The test `hasEqResidWith` will fail, since no equality constraint yet exists between the second two ψ-terms. Next, a new residuation will be created, attached to the R3 ψ-term, and assigned to `ry`. The best known sort computed for these ψ-terms will be `bc`. However, when the `setsort` routine is executed for `rx`, it will find another residuation already attached, thus it will check all the attached residuations to make sure that the new sort, `bc`, does not cause any disentailment. The structures that result at this point in the execution are shown in Figure 17.

After the new `EqResidInfo` structures have been added `canUnify` is called recursively on each pair of ψ-term that can be reached by the features in common to both of the original ψ-terms. In this case the only common feature is *x*. Since there is no equality constraint (for this frame) between R2$\rightarrow$*x* and R3$\rightarrow$*x* the best sort will be computed, in this case, it results in failure, since $ab \cap c$ is bottom.

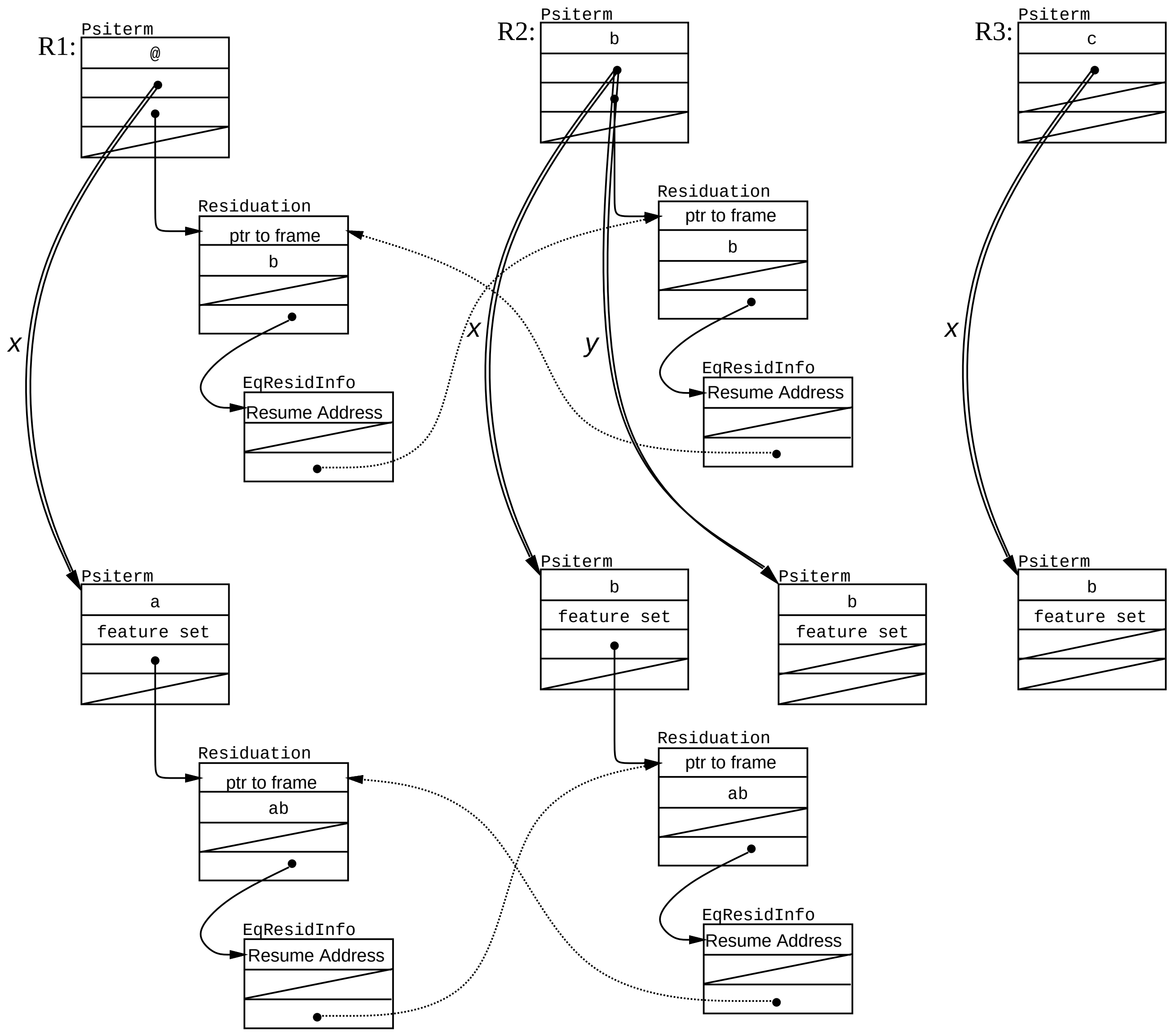


Figure 16: *ψ-terms and their associated residuations after the execution of the first equality constraint. Double lines represent features. Single lines pointers. Dotted lines pointers from one EqResidInfo to another.*

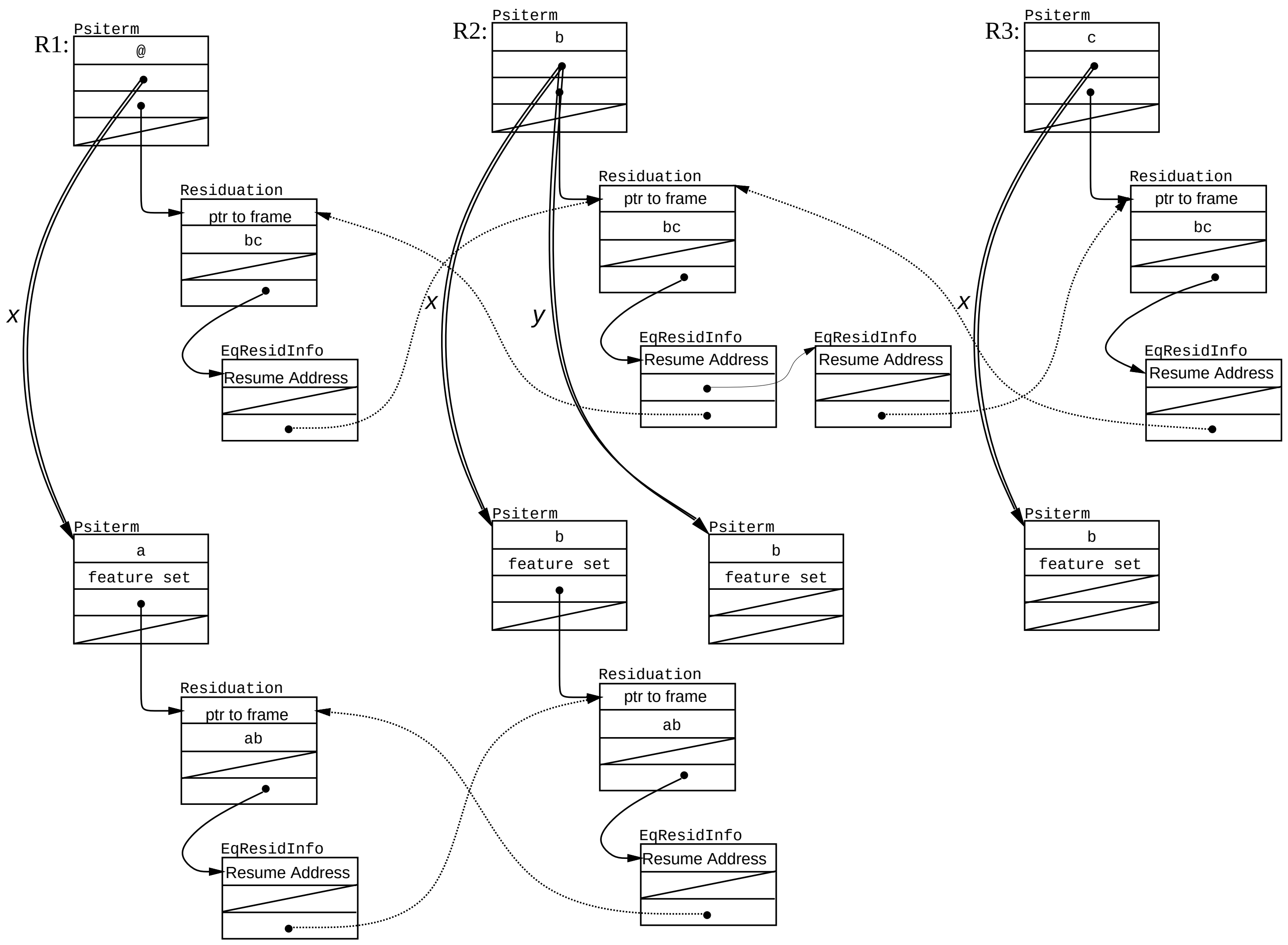


Figure 17: *The intermediate structures built during the execution of the second constraint.*

10.2.4 Initialized Constraint

See Page 20 for the pseudo-code definition of this constraint.

```
void
initialized(Psiterm& r,          // psiterm we want are checking
            Code label,          // address to goto on resumption
            Code skip)           // address to continue at on residuation
{
    // This is a funny constraint which is inserted by the compiler so that equality
    // constraints can be cleaner. It is called with a pointer to a psiterm in the frame.
    // If the frame slot has not been filled in, because a subtree was never executed due
    // to some other constraint residuating, then the slot will be NULL. If the slot is
    // NULL, then we create a psiterm in the slot with TOP as its sort and residuate.

    if (r) CONTINUE;

    // Create a new dummy psiterm. It will get unified with the psiterm inserted into
    // the frame slot which will kick off this computation.

    r = new Psiterm(TOP);
    r.tryAddSimple(CF, TOP, label);
    SKIP_TO(skip);
}
```

10.2.5 Check Residuation Count Instruction

See Page 23 for the pseudo-code definition of this constraint.

```
void
residQ(void)
{
    // Check to see if this frame has any residuation. If it does, then we return from
    // this frame's context and continue at the point after the call to this function was
    // made.

    if (CF->hasResids()) RETURN;

    // Let's execute the function.

    CF->execute();
}
```

10.3 Auxiliary Functions and Class Implementations

```
void
Psiterm::tryAddSimple(Frame *f,
                      Sort s,
                      Code label)
{
    Residuation* resid;

    resid = findResid(CF);          // Get the residuation for the current frame (CF). It
                                    // will be created if necessary.

     // Make sure that the residuation that is already attached to this psiterm doesn't
     // have a sort conflicting with s. Also update the sort in the residuation to reflect
     // the fact that this psiterm will finally end up with a sort which is <= to s.

    if (!resid->compatibleWith(s)) FAIL;

    // everything is ok so far, so add 'label' to the list of addresses to be resumed
    // when the psiterm is lowered.

    resid->addSimpleResid(label);
}

// lower is called only if there are any residuations attached to this Psiterm and the
// Psiterm has been "lowered".

void
Psiterm::lower()
{
    Residuation* r;

    // save current environment on stack

    push(CPT);
    push(CF);

    // match down rlist, invoking every ResidInfo that we encounter.

    CPT = this;                         // set the CPT register to point to this Psiterm.
    foreach (r, rlist)
    {
        r->lower();
    }

    // restore current environment

    pop(CF);
    pop(CPT);
}

// The lower() routine for class Residuation will activate all the rinfo's for the frame
// of this residuation. It assumes that the CPT register has been setup.

void
Residuation::lower()
{
    ResidInfo* todo;

    CF = parent;                        // establish the current frame

    foreach (t, todo)
    {
        t->resume();                    // execute this ResidInfo
    }

    // now lets see if we can execute the function?

    if (!CF->hasResids()) CF->execute();
}

void
ResidInfo::resume()
{
    SKIP_TO(address);
}

void
EqResid::resume()
{
    if (CPT == other)
    {
        CF->decrResiduation();
        SKIP_TO(address);
    }
    else
    {
        // just like a call to canUnify, but the book-keeping is different, in that it
        // doesn't re-create the toplevel residuations, since they already exist.
        canReUnify(CPT, other);
    }
}

void Frame::execute()
{
    // This function is only called when there are no residuated terms.
    assert(residCounter == 0);

    // we have committed to a particular definition of the function. A failure now
    // resorts to the general failure/trailing mechanism.
    fail = NULL;

    // establish the result psiterm
    RR = result;

    // start execution!
    SKIP_TO(body);
```

}

11 Conclusion

Although compiling LIFE, a very rich and powerful language, into native machine code appears daunting, LAM is a lever which makes the task feasible. This note outlines LAM and the mechanisms which give LAM the power to operate on LIFE structures easily and efficiently.

It has been our experience that getting the architecture correct with respect to residuation and entailment has not been difficult. However, getting disentailment to work correctly has proved elusive. In particular, equality constraints pose a hard problem. LAM handles the case of an equality constraint in the function head with minimal overhead. However, we need to extend the model slightly to handle special cases of equality introduced by the function call.

In spite of this shortcoming, LAM has led to valuable insights about LIFE. It has pointed the way towards the correct handling of lazy unification of order-sorted-feature terms[3]. In addition, by showing a possible approach to successfully compiling away the overhead of matching and residuation in LIFE, it should be a good starting point to the creation of the actual LIFE compiler.

A Instruction Summary

Rx$\subseteq s$	Subsort Constraint
Rx$\subseteq s$ *recheckadr*	
Rx$\subseteq s$ *failadr*, *residadr*	
Rx.ℓ?	Feature Existence
Rx.ℓ? *residadr*	
Rx.ℓ? *residadr*, *recheckadr*	
Rx $\doteq$ Ry?	Equality Constraint
Rx $\doteq$ Ry? *recheckadr*	
X?	Initialized Constraint
X? *residadr*, *recheckadr*	
Rx.$\{\ell_1, \ell_2, \cdots, \ell_n\}$	Arity Constraint
resid?	Residuation Check
resid? *residadr*	
Rx:s	Unify with Sort
Rx $\leftarrow$ Ry.ℓ	Fetch Feature
addfeature Rx, ℓ, Ry	Feature Creation
Rx $\doteq$ Ry	General Unify
unify Rx, Ry	
ref Rx, Ry	Create Reference Link
deref Rx, Ry	Retrieve a Dereferenced ψ-term
eval Rx	Evaluate by Normalization
evalfunc Rx	
evalpsi Rx	
new Psi	Heap Allocation
new Psi(s)	
new Frame$(name)$	
new SortResid$(recheckadr$, Rx, Ry$)$	

References

1. Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. Research Report 13, Digital Equipment Corporation, Paris Research Laboratory (June 1991). Revised November 1992.

2. Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1991).

3. Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. Order-sorted feature theory unification. (1992).

4. Gerard Ellis and Peter Van Roy. Constraints + control = compilation, or how to compile matching and residuation in LIFE. draft (1991).

5. A. Mariën and B. Demoen. A new scheme for unification in wam. In *1991 International Symposium on Logic Programming*, pages 257–271. MIT Press (Oct. 1991).

6. M. Meier. Compilation of compound terms in prolog. In *North American Conference on Logic Programming*, pages 63–79. MIT Press (Oct. 1990).

PRL Technical Notes

The following documents may be ordered by regular mail from:

Librarian – Technical Notes
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92563 Rueil-Malmaison Cedex
France.

It is also possible to obtain them by electronic mail. For more information, send a message whose subject line is `help` to `doc-server@prl.dec.com` or, from within Digital, to `decprl::doc-server`.

Technical Note 1: *Wild-LIFE, a User Manual*. Hassan Aït-Kaci and Richard Meyer. (being revised).

Technical Note 2: *Wild-LIFE, an Implementation Manual*. Richard Meyer. (being revised).

Technical Note 3: *Characterising Perle0*. Alan Skea. October 1990.

Technical Note 4: *Perle1DC: a C++ Library for the Simulation and Generation of DECPeRLe-1 Designs*. Hervè Touati. February 1994.

Technical Note 5: *TiGeR Version 1.0 User Guide*. Olivier Coudert, Jean-Christophe Madre, and Hervè Touati. January 1994.

Technical Note 6: *Tgr Version 1.0 Reference Manual*. Olivier Coudert, Jean-Christophe Madre, and Hervé Touati. August 1993.

Technical Note 7: *Compiling Order-Sorted Feature Term Unification*. Hassan Aït-Kaci and Roberto Di Cosmo. December 1993.

Technical Note 8: *Compiling LIFE*. Richard Meyer. December 1993.

Technical Note 9: *Riviera Class Library Reference Manual*. Didier Martineau and Thierry Pudet. May 13 1994.

Technical Note 10: *Riviera Data Library Reference Manual*. Jérôme Barraquand, Didier Martineau and Thierry Pudet. May 10 1994.

Technical Note 11: *Riviera Pricer Library Reference Manual*. Jérôme Barraquand, Didier Martineau and Thierry Pudet. Apr 21 1994.

Technical Note 12: *Riviera Wrapper Library Reference Manual*. Didier Martineau, Thierry Pudet. Apr 14 1994.

Technical Note 13: *Riviera Documentation Extractor Reference Manual*. Didier Martineau. Sep 15 1993.

Technical Note 14: *Riviera Mathematical Library Reference Manual*. Didier Martineau. Mar 28 1994.

Technical Note 15: *Riviera Visualization Tool Reference Manual*. Didier Martineau. Fev 18 1994.

Technical Note 16: *Riviera Utility Library Reference Manual*. Thierry Pudet. May 13 1994.

Technical Note 17: *Riviera Leak Library Reference Manual*. Thierry Pudet. May 13 1994.

Technical Note 18: *An Abstract Machine to Implement Functions in LIFE*. Seth Copen Goldstein. December 1992.

18

An Abstract Machine to Implement Functions in LIFE

Seth Copen Goldstein

digital

PARIS RESEARCH LABORATORY

85, Avenue Victor Hugo
92563 RUEIL MALMAISON CEDEX
FRANCE